Indoor Location – Platform for Developers and Mobile Applications

AUTHOR: Francisco José Rois Siso
DIRECTOR: Carol Davids
CODIRECTOR: Jordi Casademont Serra

Chicago, July 2017

Abstract

This paper describes the design, development, deployment and publication of the BOSSA Platform (Bluetooth and Sensors Array).
BOSSA is integrated in the Indoor Location System (IIT RTC Lab) as a central platform that serves the components of the system, and allows external developers to access the system's resources through the use of APIs.
With the inclusion of second-generation beacons, which integrate temperature and humidity sensors, the Indoor Location System increases its scope. In order to study the new possibilities, BOSSA provides support to the sensors array. In addition, the platform performs the role of Location Server for the Indoor Location System, serves tools for maintenance and test, and acts as intermediary between the database and the rest of the system. Furthermore, the project aims to promote modularity and independence between processes, and standardizes operations and internal interactions. Finally, BOSSA offers an environment for online publication of documentation and other resources concerning the project.

Resumen

Esta memoria describe el diseño, desarrollo, despliegue y publicación de la plataforma BOSSA (Bluetooth and Sensors Array).
BOSSA se integra en el Indoor Location System (IIT RTC Lab) como una plataforma central que da servicio a los elementos que componen el sistema, y permite a desarrolladores externos acceder a los recursos del mismo mediante el uso de APIs.
Con la incorporación de beacons de segunda generación, que incorporan sensores de humedad y temperatura, el Indoor Location System aumenta su potencial. Para investigar las nuevas posibilidades, BOSSA da soporte a la nueva matriz de sensores. Además, la plataforma realiza las funciones del servidor de localización para el Indoor Location System, da servicio a herramientas de mantenimiento y experimentación, y ejerce de intermediario entre la base de datos y el resto de elementos del sistema. Asimismo, el proyecto pretende promover la modularidad e independencia de procesos, y estandariza operaciones e interacciones internas. Finalmente, BOSSA ofrece un entorno para la publicación online de documentación y otros recursos asociados al proyecto.

Acknowledgements

I would like to thank Carol and Cary Davids for their energy and dedication to the work developed in the IIT Real-Time Communications Lab. They make the laboratory a place where work is enjoyable.

Special thanks to Carol for her insatiable enthusiasm, understanding and professionalism.

Finally, I want to express my gratitude to all the lab colleagues I had the opportunity to work with and collaborate to conduct this project, with special mention to Enzo Piacenza and Nathan Sowie, which were key for the development of the platform and with whom I have enjoyed working.

Table of Contents

1. Introduction

The purpose of the Indoor Location System is to provide indoor location of an emergency 9-1-1 caller who is inside a building and uses a smartphone to place the call. The system consists in a proof of concept that makes use of Bluetooth technologies. The system has been proven to work and it is consistent with the Roadmap described in “Roadmap for improving e911 location accuracy” [11]. It provides 100% accuracy to within 21 meters in all cases tested and to within 10 meters when the most accurate location algorithm is applied, which performs trilateration with least squares. The delay from the time the caller presses the emergency button to the time the call is received at the PSAP is about 3 seconds. [1]

The work described in this document continues the Indoor Location Project, defining new requirements and developing systems to fulfil them.

BOSSA Platform is the result from the analysis and detailed study of the system’s needs, which revealed the necessity of implementing the platform.
BOSSA consists in a central platform that offers APIs in order to allow external developers to access the system’s resources, and provides an environment for publishing documentation and publications concerning the Indoor Location System. The platform also performs the Location Server role for the Indoor Location System, and adapts it to accommodate the new devices and their organizational structures. Furthermore, BOSSA pretends to promote the modularity and independence of processes between the elements of the system, and standardizes operations and internal interactions. Finally, the platform has been designed to provide room for work of future developers of the Indoor Location Project, laying the foundations for the creation of new systems.

The present document describes the systems currently developed, referencing to other relevant papers written in the context of the IIT Real-Time Communications Lab. Furthermore, this document aims to serve as a reference for future developments, that will enable the Indoor Location Project to continue to evolve. Throughout this paper, work procedures are described and methods are suggested for future BOSSA developers, in order to give continuity to the project in an efficient manner.

The document starts with a brief description of the work plan for the project. Then the main technologies involved are depicted, as well as the tools and configurations for the development environment. The next section, BOSSA Platform, concentrates the bulk of the project’s development. It describes in depth the project, the platform and the sections that form it, including the systems that interact with it. The next section depicts the processes for integration and

deployment of the platform, as well as the database used. The document ends with the project's conclusions, and potential future work is suggested.

2. Work plan

The work plan is shown in Figure 1 using a GANTT diagram, which details the tasks for the project's development and the initial time estimations for the work.

The development of the project went as planned, in overall terms, and without relevant contingencies. However, the temporal order in which some of the tasks were performed needed to be altered. This mainly affected to activities concerning APIs' generation and system's integration. The main reason for these changes is the required collaborative nature of some parts of the Indoor Location Project. Systems interact with each other, which implies that some tasks need to be done in continuous contact with developers of different parts of the systems. These collaborations often imply adjustments of the initial work plan.

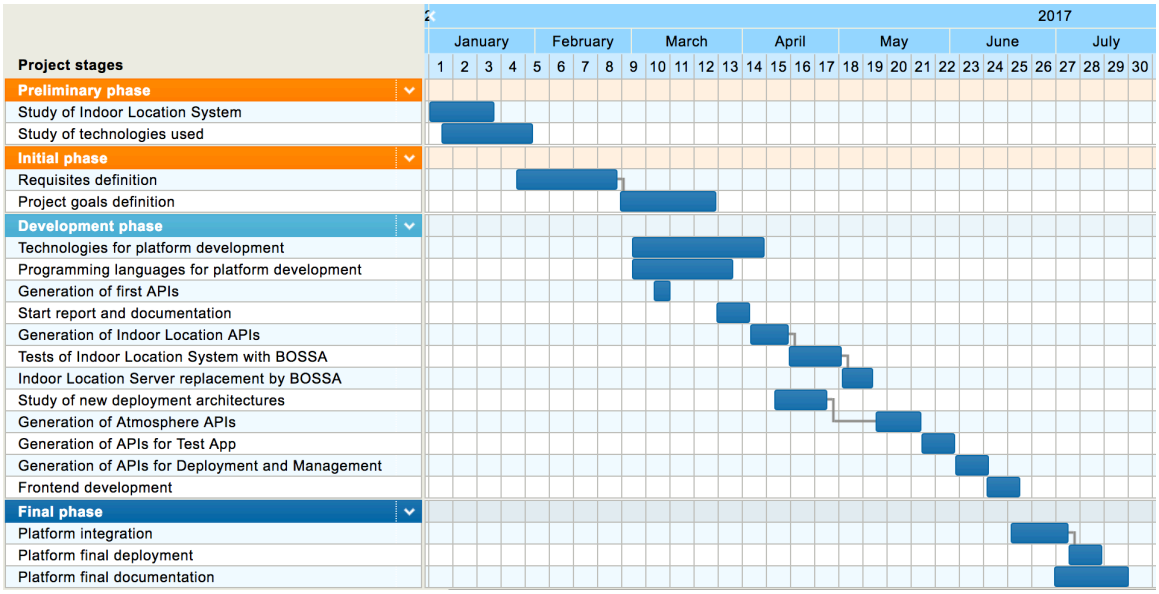

Figure 1 - Project Plan BOSSA Platform. GANTT diagram.

3. Technologies involved

This section describes the main technologies involved in the BOSSA Platform and the Indoor Location System.
First the devices for Bluetooth broadcasting and atmosphere data generation and collection are introduced, as well as the concept of API. Then Node.js, JavaScript and the Sails framework are presented, including a brief description of the MVC structure. Some remarks about Node modules are given and the relevant benefits of Android OS and MySQL database's system are shown. Finally, the main Front-End technologies used in the platform are introduced.

3.1. iBeacons and Sergeants

The Indoor Location System makes use of Bluetooth Low Energy (BLE) iBeacons. The iBeacon is a communications protocol developed by Apple and based on the BLE portion of the Bluetooth specification [16].
In previous versions of the system, the iBeacons were deployed forming an array organized following a flat architecture, in which every device performed the same advertising task, broadcasting its identification (major, minor and uuid).
The second-generation beacons include temperature and humidity sensors. In order to extract the data sampled by the sensors, a direct Bluetooth connection needs to be created.
For this reason, the second-generation beacons require the design of a new deployment architecture for buildings, in which a gateway device (sergeant) is in charge of a group of beacons (section), from which it collects data regularly. This data is then sent to a cloud service, as it is described in further sections of this document.
For more information about the deployment structure of second-generation iBeacons, see section 5.7.3 of this document.

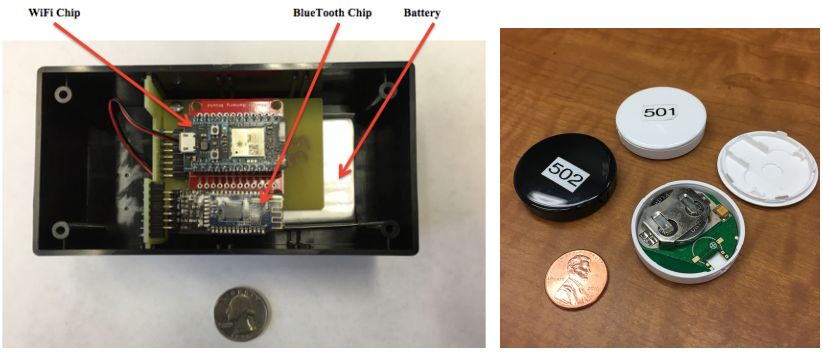

Figure 2 - BT devices: Previous beacons and new Sergeants (left). AXA beacons (right)

The picture on the left in Figure 2 shows the device that performs the role of iBeacon in the first version of the Indoor Location System, and the role of sergeant

in the system with second-generation beacons, which are shown in the picture on the right.

3.2. What is an API?

API stands for Application Programming Interface and, in computer programming, it is a set of subroutine definitions, protocols and tools for building application software. In general terms, it is a set of clearly defined methods of communication between various software components.
A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer. [19]
An API may be developed for an operating system, a database system, a web-based system, computer hardware, or software library. An API specification can take many forms, but often includes specifications for routines, data structures, object classes, variables, or remote calls. Some examples are Microsoft Windows API, Java APIs or Google APIs.
In simple terms, an API is the “messenger” that takes requests and inform a certain system of what a program wants to do, and then returns the correspondent respond back to the program.
A suitable parallelism for an API would be a waiter of a restaurant, which receives the command from a customer, takes it to the kitchen, and then returns the food (respond). In this case, the customer would be a piece of software requesting an action from a server, which is the kitchen, and the API (waiter) is the messenger between both.

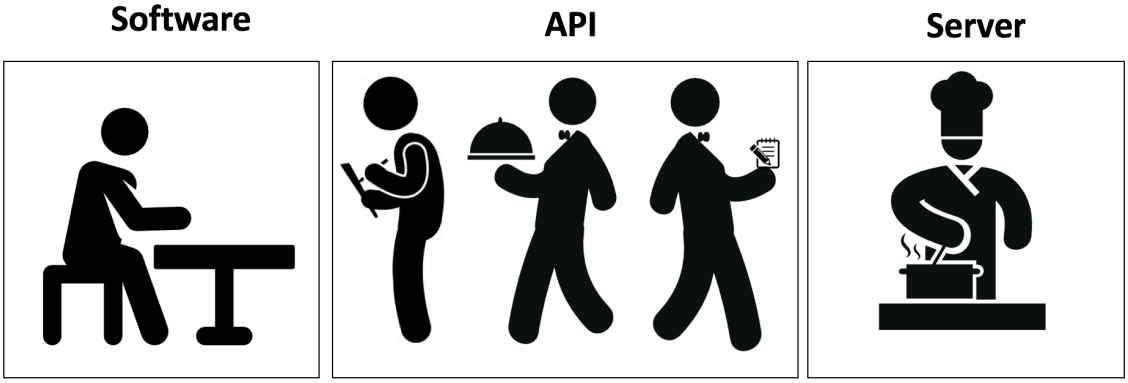

Figure 3 - Restaurant parallelism to illustrate API role

An everyday example of an application using APIs is a travel site, a website engine which looks for available flights for certain dates, by communicating with multiple external travel platforms. When a user clicks the search button on the travel site, the web server initiates a series of queries to travel platforms by using their APIs. These APIs have been developed by the different websites’ developers in order to make their resources and data available to other websites, applications and, in general, other developers. Once the information is retrieved from the several APIs,

the travel site shows the results referring to the platforms where they were found in.
Ultimately, APIs facilitate interaction between applications, data and devices. They all have APIs that allow computers to operate them and therefore, allow connectivity. [21]

3.3. Node.js / JavaScript

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.
As described in further sections of the document, Node.js' package ecosystem, NPM, is the largest ecosystem of open source code libraries in the world. [23]

Historically, JavaScript was used for client-side scripting, in which scripts written in JavaScript are embedded in a webpage's HTML. Then the user's web browser included the JavaScript engine that ran the script.
Node.js allows to use JavaScript for server-side scripting, and runs scripts on the server-side to produce dynamic web page content before the page is sent to the user's web browser.
Node.js has become one of the foundational elements of the "JavaScript everywhere paradigm", allowing web application development to unify around a single programming language.

Figure 4 - Node.js (left) and JavaScript (right) logos.

3.4. Sails.js framework

Sails is one of the most popular MVC (Model-View-Controller) frameworks for Node.js, designed to emulate the familiar MVC pattern of frameworks like Ruby on Rails, but with support for the requirements of modern apps: data-driven APIs with a scalable, service-oriented architecture. [24]

Figure 5 - Sails.js logo.

The Model-View-Controller (MVC) design pattern assign objects in an application one of the three roles: model, view, or controller. The pattern defines not only the roles objects play in the application, but also the way objects communicate with each other.
The benefits for adopting this pattern are numerous. Many objects in MVC applications tend to be more reusable, and their interfaces tend to be better defined. Applications having an MVC design are also more easily extensible than other applications. [25]

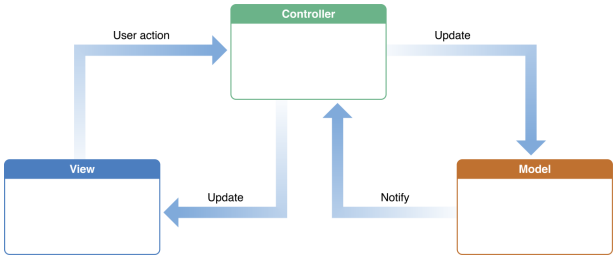

Figure 6 – MVC Flow [25]

Sails is a lightweight framework that sits on top of Express.js, which is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. [26]
Among the most relevant features of Sails.js can be found:

- **Apps written exclusively in JavaScript.**
  Sails.js is written following a convention-over-configuration philosophy. Building on top of Sails implies the applications are written entirely in JavaScript, the language already well-known to be used in the web browsers. This fact leads to more consistent styled code and more productive development.
- **Compatible with any database.**
  Using the ORM (Object-relational mapping) Waterline, it is provided a simple access layer, that makes Sails apps compatible with all of the most popular database systems. They support adapters for MySQL, MongoDB, PorstgreSQL, Redis and local disk.

- **Associations.**
  They can be assigned multiple named associations per model, different models to different databases, and joins between tables based on distinct database's technologies are supported.
- **Auto-generate REST APIs (default blueprints for development).**
  Sails offers blueprints for main backend actions by default. Generating an API ($ *sails generate api x*) implies automatic availability of methods for search, paginate, sort, filter, create, destroy, update and associate entities related to the API. These methods are the Sails blueprints and are compatible with Websockets and any supported database.
- **Declarative, reusable security policies.**
  Sails provides basic security and role-based access control by default in the form of policies, which are reusable middleware functions that run before the controllers and actions.
- **Front-end agnostic.**
  Sails is compatible with any front-end strategy: Angular, Backbone, iOS/ObjC, Android/Java, Windows Phone, and others.
- **Flexible asset pipeline (Grunt).**
  Sails uses Grunt [27], which allows customization for the front-end asset workflow. This fact also implies compatibility with Grunt modules already published. That includes support for LESS, SASS, Stylus, CoffeeScript, JST, Jade, Handlebars, Dust, and many more. When the developer decides to go into *production* mode, the assets are automatically minified and gzipped.
- **Solid foundation and low level access.**
  As described previously in this document, Sails is built on Node.js, which is a lightweight server-side technology that allows developers to write fast, scalable network applications in JavaScript. Sails uses Express.js for handling HTTP requests, and wraps socket.io for managing WebSockets. Knowing this, it is possible for developers to get into the app code and perform low-level implementations, beyond Sails high-level code. Migrating Node.js applications to Sails is also an interesting possibility.

3.5. Node.js modules

Node modules can be considered to be something similar to JavaScript libraries. They imply a set of functions that can be included in an application.
In order to use a Node module in the app's code, the module needs to be required (`var x = require ('x');`) or imported (`import x;`).

Figure 7 - NPM logo

The most used platform offering Node modules is called Node Package Manager (NPM). NPM is a Node.js packets ecosystem and the biggest open-source code libraries' ecosystem. NPM makes it easy for JavaScript developers to share and reuse code. It also simplifies the update of shared code. [33]

There are two main ways to install a module from NPM:

- Installing modules locally:
  - Manually: `$ npm install [module]`
  - Automatically: including the app as a dependency in the file `package.json`. It can be added to the file using `$ npm install [module] --save` . Then all the dependencies form `package.json` can be installed automatically by using `$ npm install` .
- Installing modules globally: `$ npm install [module] -g`

3.6. Android OS

Android OS is a well-known mobile operating system developed by Google, based on the Linux kernel and designed primarily for touchscreen mobile devices such as smartphones and tablets.
It is an open-source code and there is a huge community of Android developers that share and discuss online.
Moreover, there are many tools available to ease the process of Android apps design and development, and any app can be easily published to the Play Store to be shared with the world.
The explained are the main reasons that make Android OS the selected for the implementation of the mobile applications for the Indoor Location System.

Figure 8 – Android OS logo

3.7. MySQL databases

MySQL is an open-source Relational Database Management System (RDBMS). Its use is very extended among the developers' community and specially for web-based applications. It is used by high profile web properties including Facebook, Twitter and YouTube. [20]
Its scalability has been proven and it is compatible with the technologies used for the Indoor Location and BOSSA Platform projects.
Moreover, there are several useful tools available to work with MySQL. For this project, the tool MySQL Workbench was used.
The last reason for the use of MySQL for the Indoor Location and BOSSA Platform projects, is that previous database schemas and queries where implemented in this technology. Hence, pivoting to the use of a different database technology would imply a period of adaptation of working systems' code which, having in mind the MySQL qualities previously described, it was not considered to be a necessary process.

Figure 9 - MySQL logo

3.8. Front-end technologies

EJS (Embedded JavaScript templating) is a simple templating language that lets to generate HTML markup with plain JavaScript. [31]

It shows a similar syntax than html. This fact is different from other template engines such as Pug (previously Jade), which uses indents instead of tags, and may be confusing.

Figure 10 - EJS. Embedded JavaScript.

The basic rules for EJS templates' development are:

- JavaScript code between <% %> is executed.
- JavaScript code between <%= %> adds html to the result.

Concerning styles for the webpage, Bootstrap templates were used in the project. Bootstrap (from Twitter) is the most popular HTML, CSS and JavaScript framework for developing responsive, mobile first projects on the web [32].
Bootstrap can be included in different ways in a web application. It can be included by installing the npm package, or by directly downloading the css and js files and inserting them as templates in the app's code. The second alternative does not require any module installation and it was the option selected for the BOSSA Platform.

4. Development Environment

This section defines the tools used for the project, which form the development environment. Text editor and command line interface are commented, some relevant issues concerning Sails.js framework are explained, and the version control system and cloud services used are depicted.

4.1. Text editor and command line interface

Sublime Text 2 [22] is the tool selected for code edition in the project. It offers multiple automations and functionalities that ease the process of development.

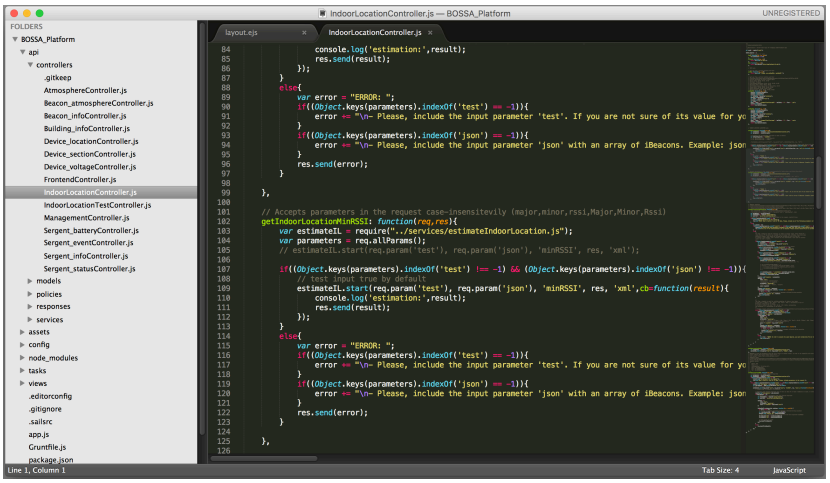

Figure 11 - Sublime Text 2. Segment from BOSSA Platform's code.

On the other hand, Terminal was used, which is the well-known terminal emulator included in the Mac OS. As a terminal emulator, the application allows text-based access to the operating system by providing a command line interface to the operating system when used in conjunction with a Unix shell, such as bash.

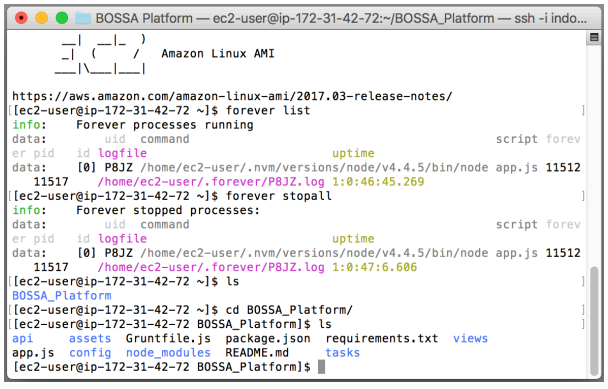

Figure 12 - Terminal

4.2. Sails.js. Main commands and tools.

These are the basic commands for Sails.js:

$ npm install sails -g : install Sails globally.
$ sails new [app] : create new Sails application.
$ sails generate api [api] : create basic controller and model for a new API.
$ sails lift OR $ node app.js OR $ start app.js : initiate the Sails app (by default on http://localhost:1337).

To learn more about Sails, follow the tutorial by the creator. You can find a link to the course in reference [10].

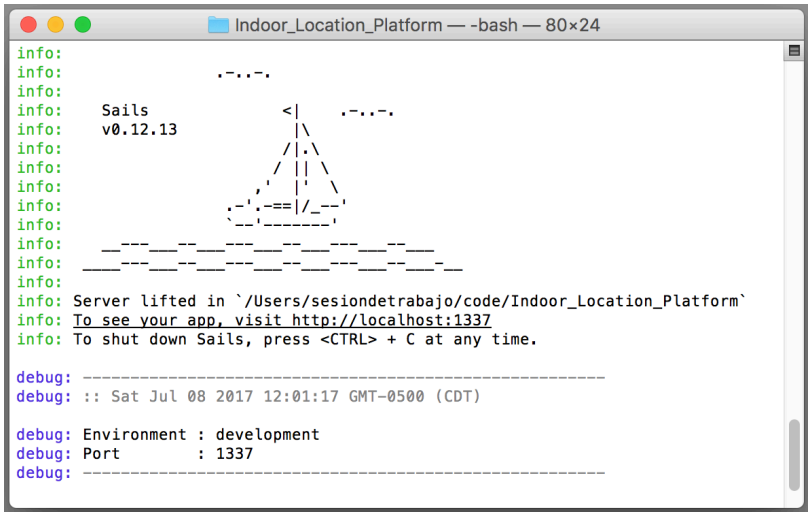

Figure 13 – Sails app lifted. Initial message in Terminal.

There are two main utilities used for the project that worth to be introduced:

- **Nodemon** : allows to re-lift the application to localhost every time that a change is detected in the code's app. It is a useful utility to improve the efficiency of the development process.
  The utility can be installed globally using $ npm install nodemon –g
  To start it, simply use: $ nodemon
- **Forever** : once the app is deployed, for example in an AWS Instance, this utility allows to run the application uninterruptedly. The commands for this utility are described in the section of the document concerning the application's deployment.

4.3. Git/GitHub

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. [29]
GitHub is a web-based Git or version control repository and Internet hosting service. It is mostly used for code. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project. [30]

Figure 14 - Git and GitHub logos

The code for the BOSSA Platform is in the git repository *BOSSA_Platform,* which is in IIT-RTC-Lab (https://github.com/IIT-RTC-Lab/BOSSA_Platform). The repo requires special permission from the administrators.
For future BOSSA developers, it is strongly recommended to continue working in the repository leaving a trace of every change, by making commented commits. Please, look at commits in the repository in order to continue the style.

General commands used to push new code:

```
$ git add .
$ git commit –am “Commit comment”
$ git push OR $ git push –u origin master
```

The repository is currently private. If it is decided to make it public, some security issues must be solved (including the concealment of authentication data, such as database access information for the models and APIs’ controllers).
During the development process of the BOSSA Platform, the git repository was linked to Heroku cloud services (in particular to the *smith-system-f* app) in order to generate automatic deployments and therefore to accelerate the testing processes. As described in subsequent sections, Heroku platform offers a simple procedure of configuration for automatic deployments, which increases the efficiency and speed of the process during development. However, the final deployment is performed in the correspondent Amazon Web Services (AWS) instance, as described in subsequent sections of this document.

4.4. Cloud services

During the development phase of the project Heroku was used, linked to the git repository. Amazon Web Services were used for final deployment of the platform.

4.4.1. Heroku

Heroku is a cloud platform based on a managed container (smart dynos) system, with integrated data services and a powerful ecosystem, for deploying and running modern apps. [28]
It simplifies the deployment process by allowing, with simple configurations, the link between a git repository and the cloud server. Therefore, every time that the developer pushes the code to the git repository, the new code is automatically deployed. This fact makes the development-testing process more quick and efficient.

Figure 15 - Logos of cloud services used for the platform. Heroku and AWS.

4.4.2. Amazon Web Services (AWS)

These are the AWS utilities employed during the project's development:

- Amazon EC2: cloud hosting service that provide resizable virtual servers.
- Amazon RDS: to set relational databases in the cloud.
- Amazon Route 53: scalable cloud DNS web service.

5. BOSSA Platform

With the inclusion of second-generation beacons, which integrate temperature and humidity sensors, the Indoor Location Project increases its scope. It is needed to restructure the system and to include new elements to it, in order to provide support to the new devices and their organizational architectures.
In addition, it is required to “open” the system and to make the resources available for external developers. To do so, creation, documentation and publication of APIs is needed. The APIs also need to perform the indoor location functions.
Furthermore, the increasing complexity of the system makes it require more than ever modularity and independence between the elements of the system. Thus, standardization for internal interactions is needed, as well as for development, update, deployment and publication of new systems for the Indoor Location Project.
Finally, it is a requirement to redesign the indoor location database in order to adapt it to the new nature of the Indoor Location System, and to have space for future developments.

BOSSA is the platform developed in this project for the purpose of satisfying all the requirements described, integrated in the new system as a central platform that interacts with all the elements, mainly using APIs.

In the following subsections of the document the Indoor Location System’s operations are described, as well as the role that the platform performs in the new system. Moreover, the requirements for the BOSSA Platform are depicted, and the code structure for the platform is shown, including definitions for the elements that form it. Finally, five different sections are defined in the platform and they are described in depth.

5.1. Indoor Location System

When a person calls an emergency number, such as 911 in the US, the location of the caller must be provided to the network. It is used for two purposes: first, to route the call to the appropriate answering point; second, to display the caller’s location to the call-taker who will dispatch first-responders.
When the call is made from a mobile device inside a building, the call-taker and the first responders need more information than the latitude and longitude and/or the street address in order to find the caller. They need accurate indoor location, which can be for example a floor and room number.
The Indoor Location System is a proof of concept system that provides the floor and room number as well as the street address of a caller. A method for providing the (x,y) coordinates of the caller on the correct floor has also been created an is described in subsequent subsections of the document.

The proof of concept system was developed in response to the “Roadmap for Improving E911 Location Accuracy” [11] developed by an alliance of the Association of Public- Safety Communications Officials-International (APCO) [12], the National Emergency Numbers Association (NENA) [13], and four national wireless carriers, AT&T, Sprint, T-Mobile and Verizon.
The Indoor Location System is integrated with a replica of an Emergency Services IP Backbone Network (ESInet)] on a test-bed in the IIT Real-Time Communications Lab [14]. The ESInet is specified in the NENA i3 Standard [15]. The roadmap called for the use of Wifi Access Points, Bluetooth Beacons or both to be the source of the indoor
location. A proof of concept system that used the Wifi Access Points on the University campus was developed and is described in [9]. The current work makes use of Bluetooth Low Energy (BLE) iBeacons, leveraging the experience gained in that earlier work. The iBeacon is a communications protocol developed by Apple and based on the Bluetooth Low Energy portion of the Bluetooth Specification. Development of a new configuration for the array, that uses a hierarchical rather than a flat architecture is in progress, and described in subsequent sections of this document. The iBeacons in the new array are divided into groups, each of which is managed by a gateway device (sergeant). The new architecture is described in section 5.7.3 of this document.

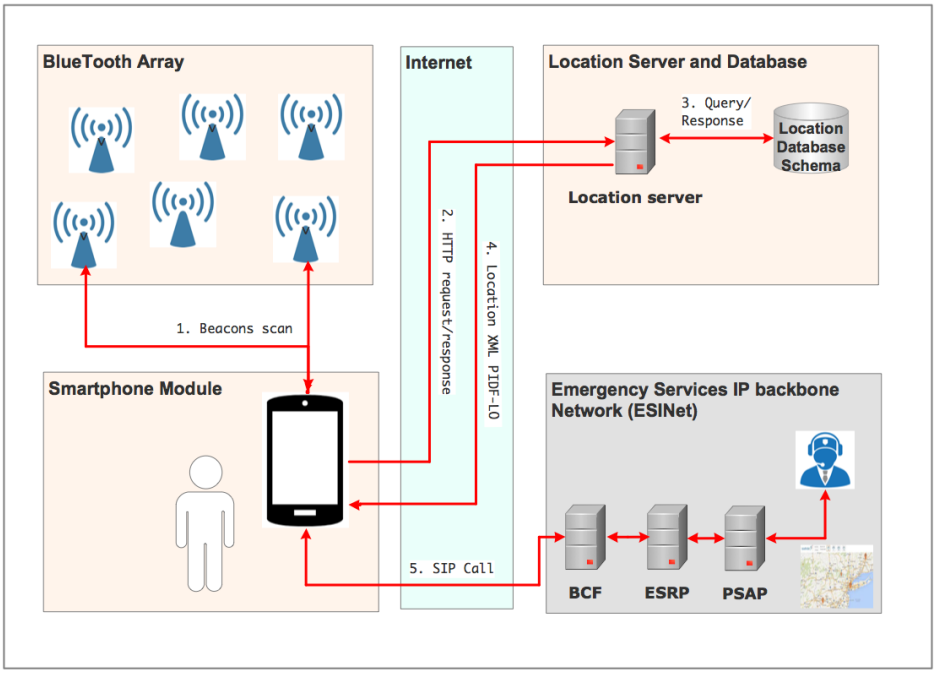

Figure 16 - Indoor Location System. Before BOSSA Platform.

The system consists of the following four components: (1) A smartphone application that makes the emergency call, obtains the location of the caller, and forwards the call, including location information, to the answering point; (2) An array of Bluetooth Low Energy (BLE) devices that are deployed on the inside walls of a building at selected locations; (3) A database that contains the location of the BLE devices; (4) A Location Server capable of querying the database to discover the location of particular BLE devices, and of using the location information thus obtained to calculate the location of the caller, format this location, and send the formatted information to the smartphone. Figure 16 illustrates these components and the flow of information between them. In the Proof of Concept System, the location server and database are local, whereas in the Roadmap [11] it is centrally located and available to callers in all compliant buildings. [1]

When an emergency call is placed using the smartphone with the Caller App installed, first a Bluetooth scan for beacons is performed. Then, the data collected is sent to the Location server module, which executes a location algorithm and consults the database to determine the location of the caller, and sends the formatted results to the phone application. Finally, the indoor location is inserted in the body of the SIP message that establishes the call with the ESInet. The location of the caller is displayed on the screen of the emergency call taker.
The call process for the Caller App is detailed in section 5.5.3, and the indoor location algorithms are described in section 5.5.6 of this document.

The diagram in Figure 17 shows the new configuration of the Indoor Location System, with the BOSSA Platform integrated into it.
The BOSSA Platform offers APIs to determine and calculate the indoor location of a caller. BOSSA's APIs for indoor location receive data about beacons, query the database to know the deployment location of them and, applying location algorithms, they determine where is the smartphone, and therefore also the caller. The location information is then sent to the smartphone properly formatted and ready to be sent to the ESInet or the correspondent emergency calls service.
Thus, it can be said that the BOSSA Platform now performs the functions of the previous Location Server, and therefore it executes the Location Server's role for the Indoor Location System.

BOSSA's APIs for Indoor Location and the location algorithms are described in section 5.5 of this document.

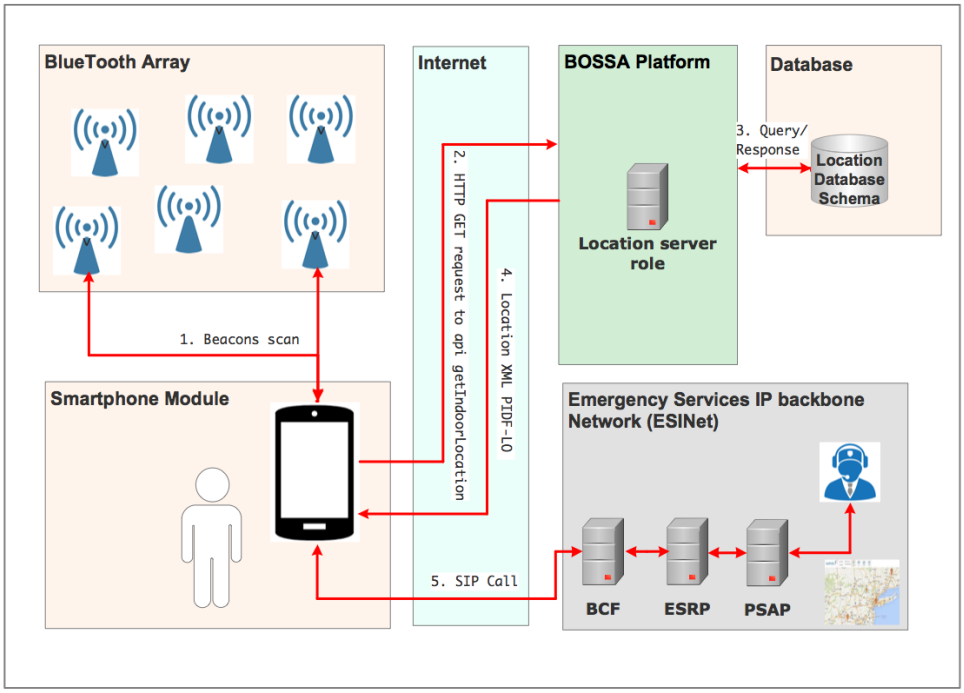

Figure 17 - Indoor Location System. After BOSSA Platform.

5.2. Purpose and requirements for the BOSSA Platform

This section describes the purpose and requirements for the BOSSA Platform:

1. **Access for external developers.**

Offer to developers a uniform and comprehensive list of APIs that they can use to retrieve data and perform actions in the Indoor Location System.
The APIs provide an open and standard way to interact with the platform, in such a way that developers can easily take advantage of the system's functionalities and data by simply including our APIs in their applications.

2. **Temperature and humidity data retrieval.**

The second-generation iBeacons (AXA beacons), which are used in recent deployments, are able not only to provide Bluetooth advertisement functionalities but also temperature and humidity data samples. They include sensors to do the atmosphere measurements.
The platform needs to be able to automatically collect information from the iBeacons and store it in the correct database. It also needs to provide APIs for external pieces of software to retrieve this information. These processes and the solutions offered by the platform are described in subsequent sections of this document.

The atmosphere (temperature and humidity) data could be used, for example, to generate heat maps for buildings, to learn about atmosphere data in rooms in real time, or to do analytics using the historical of humidity and temperature data for a certain period of time.
These functionalities could be use by emergency services, firemen departments, or by HVAC systems' technicians for room atmosphere control.

3. **Indoor Location Server integration.**

The BOSSA Platform integrates the indoor location functionalities for execution of location algorithms and generation of formatted data. It includes APIs for this purpose, and other APIs to fulfill the requirements of the elements of the system.
Therefore, it can be said that the platform performs the Location Server's Role for the Indoor Location System.

4. **Unification of processes.**

Previously to the BOSSA Platform, some tasks were performed in different ways depending on the software that was performing the task. For example, access to databases was done differently for separated pieces of software. The BOSSA Platform pretends to condense all the processes, so they all are performed by using the APIs, and therefore done in a central platform and in a uniform way.

5. **Modularity.**

As the system evolves and its scope widens, the need of uniform interaction between independent software pieces gains importance. Including APIs on the central platform allows the rest of the modules of the system (including the phone applications) to refer to it as a "black box" that receives queries and answers the requested information or a confirmation of action. It is intended that errors are isolated in different modules of the big project, so troubleshooting efficiency increases.

6. **Standardization, control and security.**

The system allows other pieces of software, devices or users to perform actions in the system or to retrieve information from it. These functions are provided by the BOSSA Platform in a controlled and standard way to every interactor (internal or external), being the platform always an intermediary between them and other modules such as the databases.
The APIs follow common standards and therefore their use intends to be simple and intuitive. Furthermore, every task performed in the system, especially the ones implying database insertions or other modifications, go through the platform and are controlled by it. Unwanted intrusions are denied and therefore security is also improved.
Furthermore, the technologies employed allow the implementation of the platform using a single programming language (JavaScript), and to generate Back-end and Front-end integrated in the same application. These facts can imply higher efficiency for programming and for code execution flow, and may facilitate the development process and app security.

5.3. Platform code structure and descriptions

The web application BOSSA_Platform follows the usual structure of Node.js applications developed using the Sails.js framework.
In this section, the most relevant folders and files from the code are discussed, as well as their main configurations and implementations in BOSSA.
The reader should pay special attention to the controllers, views and dependencies, which currently concentrate most of the platform's logic, front-end code and Node modules, respectively.

- **package.json:** it includes dependencies and basic metadata about the project. When an app is set in a new device, using the command *$npm install* will automatically install all the required modules, which are specified in the file package.json. During development, in order to properly include the module in the list of dependencies, the developer should include the option *--save* when installing it.
  In the following, the main node modules included in the platform are described:
  - async: provides straight-forward functions for working with asynchronous JavaScript.
  - ejs: Embedded JavaScript templates.
  - grunt (and other Grunt-related modules): JavaScript Task Runner.
  - mysql: node.js driver for MySQL databases.
  - python-shell: allows to run Python scripts from Node.js with basic but efficient inter-process communication and good error handling. Only needed for the BOSSA Platform temporarily, until some of the location algorithms are translated from Python to JavaScript language.
  - rc: non-configurable configuration loader.
  - sails: main Sails.js module.
  - sails-disk: local disk adapter for the Sails framework and Waterline ORM. Bundled by default in new Sails projects.
  - sails-mysql: MySQL adapter for the Sails framework and Waterline ORM. Allows to use MySQL via the models to store and retrieve data. Also provides a query() method for a direct interface to execute raw SQL commands.
- **app.js:** it is the entry point of the web app. This file is what is running once the application is deployed, but not while in development mode (using localhost:1337). The JavaScript file lifts the sails app programmatically.
- **.editorconfig:** provides simple way to specify tabs, spaces and other related configurations.
- **.gitignore:** informs git of what to keep and what to ignore when the code is pushed to the git repository.

- **README.md:** convention of GitHub. It usually includes instructions to launch or initially configure the app.
- **Gruntfile.js:** entry file for Grunt.
- **sailsrc:** permits to perform high level configurations. It has priority over other configurations in the application.
- **api folder:**
    - **Models:** provide the structure for the records and attributes of the APIs, which are directly related to entities from the database. They need a controller attached to each in order to conform a functional API.
      The BOSSA Platform code includes one model for each of the entities in the main database (*indoor_location_db*):
        - **Beacon_atmosphere.js**
        - **Beacon_info.js**
        - **Building_info.js**
        - **Device_location.js**
        - **Device_section.js**
        - **Device_voltage.js**
        - **Sergent_battery.js**
        - **Sergent_event.js**
        - **Sergent_info.js**
        - **Sergent_status.js**

      Moreover, there are other models related to Indoor Location, Atmosphere, Management and Frontend controllers, that implement the functional APIs of the platform. They are not linked to any entity in the database. Therefore, these models are not used in practice for more than completing the API concept.

    - **Controllers:** comprise the logic to define the behavior of the APIs. Behind the scenes and during development, sails provides five standard actions: find, findOne, create, update and destroy. They facilitate the test and development process.
      The BOSSA Platform code, in parallel to the models described above, includes, on the one hand, controller specifically designed to make use of the models related to the entities in the database (*indoor_location_db*):
        - **Beacon_atmosphereController.js**
        - **Beacon_infoController.js**
        - **Building_infoController.js**
        - **Device_locationController.js**
        - **Device_sectionController.js**

- **Device_voltageController.js**
- **Sergent_batteryController.js**
- **Sergent_eventController.js**
- **Sergent_infoController.js**
- **Sergent_statusController.js**

On the other hand, several controller JavaScript files implement the logic for the Platform's APIs. The APIs' purpose and implementation are described in subsequent sections of this document:

- **AtmosphereController.js:** APIs for atmosphere (temperature and humidity) data retrieval.
- **IndoorLocationController.js:** APIs for execution of Indoor Location algorithms and retrieval of indoor maps.
- **IndoorLocationTestController.js:** APIs for Indoor Location testing purposes and algorithms' improvement. These APIs are used by the Test App (mobile application) in order to receive information about the available testing points and locations, generate test events and store the resultant test information in the analysis database *(testing_application_db*).
- **ManagementController.js:** APIs for iBeacons' deployment and management. This includes installation of devices in new buildings, replacement of already deployed devices and update of any other devices' related information in the main database (indoor_location_db).
- **FrontEnd.js:** APIs to serve the requirements of the frontend resources in the Node application.

- **policies:** provide a way to protect the actions performed in the app. It allows to keep the access control logic separated from the rest of the code.
- **responses:** functions that can be called from the control code in order to send a standard final respond. They are terminal functions. They can be used in the code by using *res.[function]*. When the response to be sent is not clear, *res.negotiate* can be used and the *best guess response* will be sent.
- **services:** the resources included in this folder are available throw out the app. In the particular case of the BOSSA Platform app, it includes scripts for the estimation of the indoor location, by using the correspondent algorithm.

- **assets**: this folder is a pool of all the assets needed by the application, including images, scripts and files for format and style configurations. The folder **Images** contains a subfolder named FloorPlans, containing maps of the buildings where the Indoor Location system is currently deployed.

The folder **py** contains the Python scripts needed for *LeastSquared* indoor location algorithm. The correspondent script is called by an API from the IndoorLocation APIs' section. Currently (July 2017) the python scripts (*gettests.py* and *py_script_LeastSquared.py*) use the Python libraries *lmfit, math, numpy, sys, argparse, json, csv*, records and *PrettyTable*. These libraries need to be installed when the platform is set in a new machine. The folder **styles** includes the style files for Bootstrap, which is used by the application Front-end (described in subsequent sections of the document).

- **config**: as well as the APIs' controllers and models, the configuration folder is part of the application's back-end. It implies declarative instructions for app configurations.
    - **env:** allows to perform environment configurations for development and production modes.
    - **blueprints.js:** allows configuration of blueprints in the sails application in development. In production, these actions are automatically disabled.
    - **bootstrap.js:** it runs when the server is lifted. It is important to include the callback (cb) or the server will never start.
    - **connections.js:** allows to configure adapters.
      Currently the adapters configured are the *localDiskDb*, which is the default adapter for storage, and the *MysqlServer*, which contains the authentication information required to establish the connection with the main database (*indoor_location_db*).
    - **models.js:** configuration of models.
      A connection is configured to the database adapter *MysqlServer*, which is linked to the main database (*indoor_location_db*). The configuration 'migrate' is set to 'safe', which permits edition, creation and deletion of rows, but not the alteration of the database's tables. It is the adequate value for production mode. The value 'alter' can be used during development, which gives more database's configuration rights.
    - **routes.js:** configuration file for routes and mappings. Configurations with higher priority than the blueprints.
      Several routes are configured in this file, but in order to follow a general procedure to create APIs, the generation of routes by this method is avoided whenever possible. The preferred method for routing is the creation of API functions in the controllers, with the desired route name. For example, the *IndoorLocationController* provides an API named *getIndoorLocation*. This API can be automatically accessed by using the route */IndoorLocation/getIndoorLocation*, without the need to manually configure a route in *routes.js* configuration file.

    - **views.js:** allows to configure the view engine. Between the configurations are the name of the views engine, the use of a single or multiple layouts and the use of partials for the views. The engine used for BOSSA is EJS with a single layout. No partials are employed for the views of Platform.
- **node_modules:** this folder includes all the data from the node modules installed in the app.
  For more information about the modules used by the BOSSA Platform, see the beginning of this section, where the dependencies in the package.json file are described.
- **views:** views' files, including defaults, statics and layout(s).
    - **static:** main EJS files for the website, including *index.ejs* for the main screen.
    - **layout.js:** html code which allows to specify the parts of the code that will be included in every page on the web. It includes header and footer.
    - **homepage.js:** default Sails.js home page. Not used by the platform. Substituted by index.ejs.

5.4. Platform sections

The BOSSA Platform can be conceptually divided into five sections, each one of them focused on different kind of tasks.
Two of the sections are specifically related to the Indoor Location System. One for the execution of indoor location algorithms and maps retrieval: **Indoor Location algorithms and maps**. The other section addresses the analysis of the system and improvement of location algorithms: **Indoor Location testing and analysis**.
The block **Atmosphere Data** is focused on the generation, storage and retrieval of atmosphere data (temperature and humidity). The data is obtained from the second generation beacons (AXA beacons).
Deployment and Management is the section concerning the installation, substitution and removal of devices in buildings (beacons/sensors and sergeants).
The last part, **Frontend**, involves everything related to the Front-end, the publishing of the system's resources through a website, which is mainly an informative portal where documentation about the project can be found. There is also an area in the website reserved for management and deployment of devices, for which the user needs to be authenticated in order to gain access.
Figure 18 shows the complete diagram for the system. It shows the main elements that form it today (July 2017). The diagram includes mobile-phone applications, databases, internal functions and external servers.
Every element of the system is explained in detail in subsequent sections of this document, as well as the APIs relating them to the BOSSA Platform.

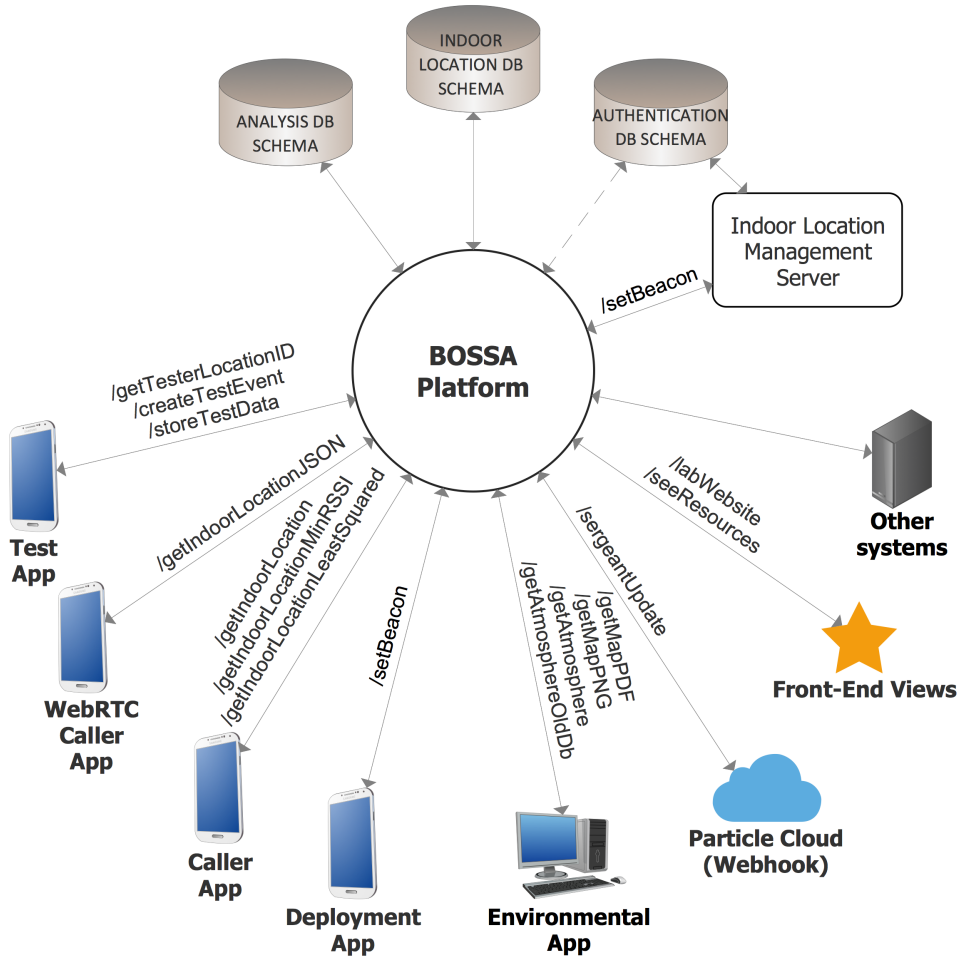

Figure 18 - Platform and interactions (APIs). Complete diagram.

In the following, each one of the elements in the diagram is briefly described and it is shown where they are running and/or where the code can be found (July 2017):

- **Test App**: Indoor Location testing and algorithm improvement.
  - *https://github.com/IIT-RTC-Lab/NG911-Bluetooth_TestApp*
- **WebRTC Caller App**: Indoor Location application for 911 emergency calls. WebRTC-based.
  - *https://github.com/IIT-RTC-Lab/WebRTC_Emergency_Service_AndroidApp*
- **Caller App**: Indoor Location application for 911 emergency calls. Use of SIP calls to ESInet.
  - *https://github.com/IIT-RTC-Lab/NG911-Bluetooth_TestApp*
  - New version: *https://github.com/IIT-RTC-Lab/NG-911-Bluetooth-2.0*

- **Deployment App**: management and deployment of devices in buildings.
  - *https://github.com/IIT-RTC-Lab/BeaconUpdateAndroid*
- **Environmental App**: atmosphere (temperature and humidity) data display.
  - *https://github.com/IIT-RTC-Lab/Environmental-App*
- **Particle Cloud (Webhook)**: cloud services that retrieve atmosphere (temperature and humidity) data from sergeants and send it to the Indoor Location database. The data is sent to the database using an API from BOSSA Platform, configured in the cloud's Webhook service for automation.
- **Front-End Views**: web frontend views, internally stored in the application.
  - *BOSSA_Platform/views*.
- **INDOOR LOCATION DB SCHEMA**: main database schema for the Indoor Location system and for atmosphere data storage.
  - DB Schema: *indoor_location_db*
  - Diagrams: *https://github.com/IIT-RTC-Lab/Indoor_location_database*
- **ANALYSIS DB SCHEMA**: database for Indoor Location experimental data storage. The stored data allows the analysis and consequent improvement of the indoor location system.
  - DB Schema: *testing_application_db*
  - Diagrams: *https://github.com/IIT-RTC-Lab/Indoor_location_database*
- **AUTHENTICATION DB SCHEMA**: database schema for authentication, needed for critical management functions of the system. Currently it is used for authentication from the *Indoor Location Management Server,* which allows to update information about the devices deployed. In the future, this database schema will include other authentication data for other sections of the platform where needed.
- **BOSSA Platform**:
  - *https://github.com/IIT-RTC-Lab/BOSSA_Platform*
  - AWS EC2 Instance (Amazon Linux): *BOSSA Platform*
  - Subdomain*: api.iitrtclab.com*
- **Indoor Location Management Server**: web application for management purposes for the Indoor Location system.
  - https://github.com/IIT-RTC-Lab/Indoor_Location_Management_Server
  - AWS EC2 Instance (Amazon Linux): *Indoor Location Management Server*
  - Subdomain: *management.iitrtclab.com*

Each one of the following sections of the document is linked to a section of the BOSSA Platform. They are explained in detail, as well as the APIs that the platform offers for each section, and the main systems that make use of the APIs. For that purpose, the diagram in Figure 18 is dissected in order to show only the relevant elements for each section

5.5. Indoor Location algorithms and maps

This section deals with location algorithms and floor maps retrieval for the Indoor Location System.
Figure 19 shows a partial diagram of the platform including database schemas and other systems involved with this section, as well as the APIs used to interact with the BOSSA Platform.
The following subsections detail the APIs available in the Platform for execution of Indoor Location algorithms and for indoor maps retrieval, as well as the generic APIs, which are linked to database entities. Furthermore, the mobile applications for emergency calls are introduced. Finally, the Indoor Location algorithms are described.

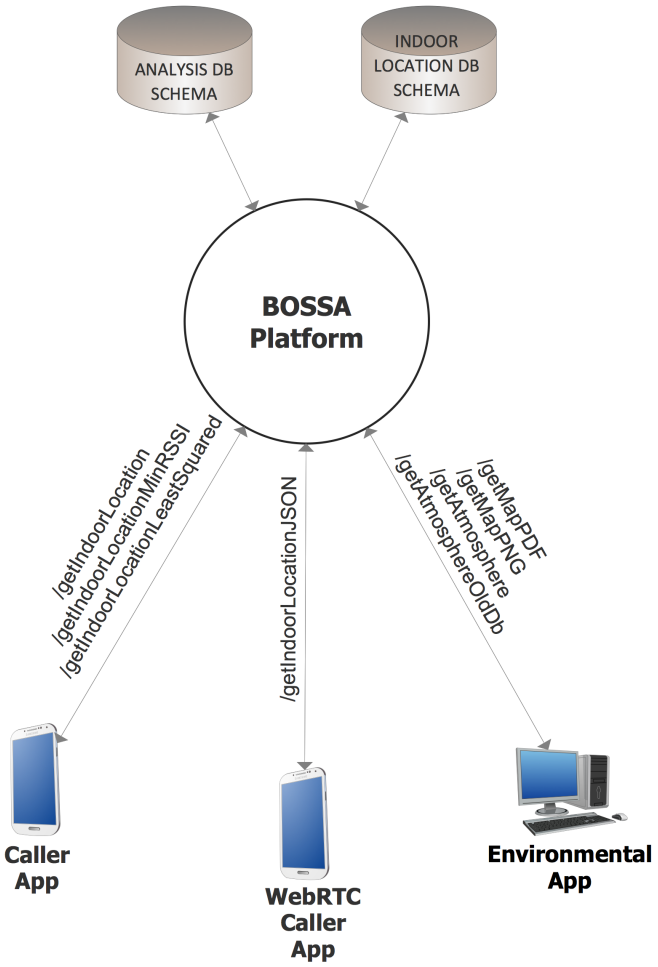

Figure 19 - Indoor Location algorithms and maps. Partial platform's diagram.

5.5.1. IndoorLocation APIs

IndoorLocation APIs deal with location algorithms execution and indoor maps retrieval.
They are implemented in the IndoorLocation controller.

5.5.1.1. APIs for Indoor Location algorithms

Currently (July 2017), the Indoor Location algorithms implemented in the platform are *minRSSI* and *Least Squared.*
The following APIs replace the main functionalities of the previous Indoor Location Server and add new ones.

- **/getIndoorLocation**?test=[test]&json=[json]

  Returns the indoor location result from the default location algorithm's execution, which can be easily changed to a different one in the code. Today (July 2017), the minRSSI algorithm is set to be the default.
  The algorithm estimates the indoor location taking as input the data from the iBeacons included in the request's json parameter.
  The result returned is in XML PIDF-LO format.
  Both parameters are mandatory in the request (test and json):
  - test: identifies test requests to the API. By default, set to true.
  - json: includes an array of iBeacons, including for each one its Major, Minor and Rssi, in JSON format.

  Example of use:
  - /getIndoorLocation?test=true&json=[{"Major":101,"Minor":175,"Rssi":-91.0},{"Major":101,"Minor":175,"Rssi":-92.0},{"Major":101,"Minor":175,"Rssi":-93.0},{"Major":101,"Minor":202,"Rssi":-80.0},{"Major":101,"Minor":202,"Rssi":-83.0}]

- **/getIndoorLocationMinRSSI**?test=[test]&json=[json]

  Returns the indoor location result from the execution of the minRSSI location algorithm.
  The algorithm estimates the indoor location taking as input the data from the iBeacons included in the request's json parameter.
  The result returned is in XML PIDF-LO format.
  Both parameters are mandatory in the request (test and json):
  - test: identifies test requests to the API. By default, set to true.
  - json: array of iBeacons, including for each one its Major, Minor and Rssi, in JSON format.

This API executes the JavaScript script *estimateIndoorLocation.js*, located in *api/services*. The script is based on a script from the previous Location Server. The script was adapted to the platform's code style, including callbacks and others, and integrated into the app's structure. The queries were implemented to access the correct tables and fields in the new Indoor Location database *(indoor_location_db*).

Example of use:

- /getIndoorLocationMinRSSI?test=true&json=[{"Major":101,"Minor":175,"Rssi":-91.0},{"Major":101,"Minor":175,"Rssi":-92.0},{"Major":101,"Minor":175,"Rssi":-93.0},{"Major":101,"Minor":202,"Rssi":-80.0},{"Major":101,"Minor":202,"Rssi":-83.0}]

- **/getIndoorLocationLeastSquared**?json=[json]
  or
  getIndoorLocationLeastSquared?testid=[testid]
  or
  getIndoorLocationLeastSquared?testid=[testid]&nbeacons=[nbeacons]&nfbeacons=[nfbeacons]&proximity=[proximity]

  Returns the x and y coordinates, result from the execution of the Least Squared algorithm.
  The algorithm calculates the coordinates taking as input the data from the iBeacons included in the request's json parameter.
  The algorithm is only executed if data from at least 4 iBeacons is specified. This is for conceptual reasons concerning the Least Squared algorithm.
  Instead of the json parameter, it can be specified the test id (testid), for analysis purposes.
  One and only one of the following input parameters must be included in the request: json OR testid. The rest of the parameters related to testid are optional.
  - json: array of iBeacons, including for each one its Major, Minor and Rssi, in JSON format.
  - testid: flag used for analysis; it takes an integer input, which should be a valid test id. It returns a json output, with information needed to render the test's results. The testid parameter accepts several parameters as options, which can be individually included or not: nbeacons, nfbeacons and proximity.

  The Least Squared algorithm is currently (July 2017) defined in the Python programming language. It is the only part of the platform's code that is not

implemented in JavaScript. Therefore, the Python scripts are executed from the API, using the Node module *python-shell*.
The Python scripts are located in *assets/py*:

- *gettests.py*: contains the code handling getting tests from the database, and packaging the data into a more convenient object form.
- *py_script_LeastSquared.py*: applies the Least Squared algorithm.

The core library used by the Python scripts is *lmfit*, which is a wrapper around "*scipy.optimize.leastsq"*, using the Levenberg-Marquardt algorithm. For future developments, it would be desirable to convert the logic of the Least Squared algorithm to JavaScript language in order to have the complete platform integrated and developed in the same programming language.

Examples of use:

- /getIndoorLocationLeastSquared?json=[{"Major":101,"Minor":175,"Rssi":-91.0},{"Major":101,"Minor":175,"Rssi":-92.0},{"Major":101,"Minor":175,"Rssi":-93.0},{"Major":101,"Minor":202,"Rssi":-80.0},{"Major":101,"Minor":202,"Rssi":-83.0}]
- /getIndoorLocationLeastSquared?testid=333
- /getIndoorLocationLeastSquared?testid=333&nbeacons=5&nfbeacons=3&proximity=-70,5,-60,3,-50,2,-40,1

- **/getIndoorLocationJSON**?test=[test]&json=[json]

Returns the indoor location information and x, y coordinates, results from the execution of the minRSSI and Least Squared location algorithms, respectively.
The algorithms estimate the indoor location and coordinates taking as input the data from the iBeacons included in the request's json parameter.
The result returned by the API is in JSON format.
Both parameters are mandatory in the request (test and json):

- test: identifies test requests to the api. By default, set to true.
- json: array of iBeacons, including for each one its Major, Minor and Rssi, in JSON format.

The two algorithms (*minRSSI* and *LeastSquared)* are executed only in the case that data from at least 4 iBeacons is included in the request. This is for a reason of concept concerning the Least Squared algorithm. If the json input parameter shows information of 3 or less beacons, only the *minRSSI*

algorithm is executed, and therefore no x, y coordinates are included in the API's result.
It is important to do the request to the API as shown in the example below, including Major, Minor and Rssi starting with capital letter. The LeastSquared algorithm is implemented to be case sensitive.

Example of use:

- /getIndoorLocationJSON?test=true&json=[{"Major":101,"Minor":175,"Rssi":-91.0},{"Major":101,"Minor":175,"Rssi":-92.0},{"Major":101,"Minor":175,"Rssi":-93.0},{"Major":101,"Minor":202,"Rssi":-80.0},{"Major":101,"Minor":202,"Rssi":-83.0}]

5.5.1.2. APIs for Maps

APIs for Maps are designed to allow the retrieval of floor maps of buildings where the Indoor Location System has been deployed.
The maps can be obtained in PDF or PNG formats.
Today (July 2017), the maps are stored internally in the BOSSA_Platform web application. They can be found in *assets/images/FloorPlans.*
It is expected that, in the future, the maps will be redesigned and updated regularly, and maps for new buildings' deployments will be stored.
Hence, for future developments of the Platform, it is recommended to store the maps in an external server, where they could be easily updated and modify, and where the Platform could find them always up to date. This could be done in a new folder inside the same AWS instance, or in a different instance.
It is also recommended to register the maps and where to find them, in the Indoor Location main database.

- **/getMapPDF**?building=[building]&floor=[floor]

  Get map from a certain building and floor in PDF format.
  It redirects to the resource in */assets/image/FloorPlans*.
  Two parameters are needed in the request: building AND floor.
  Current building and floor maps available (July 2017):
  - *Alumni_Hall: AM-00, AM-01, AM-02*
  - *IT_Tower: IT-00-21*
  - *Life_Science_Building: LS-00, LS-01, LS-02, LS-03*
  - *Perlstein_Hall: PH-00, PH-01, PH-02*
  - *Rettaliata_Engineering_Center: E1-00, E1-01, E1-02*
  - *Siegel_Hall: SH-00, SH-01, SH-02, SH-03*
  - *Stuart_Building: SB-00, SB-01, SB-02*
  - *Wishnick_Hall: WH-00, WH-01, WH-02, WH-03*

Example of use:

- */IndoorLocation/getMapPDF?building=Alumni_Hall&floor=AM-00*

- **/getMapPNG**?building=[building]&floor=[floor]

  Get map from a certain building and floor in PNG format.
  Equivalent use, procedure and resources available than for the API /getMapPDF.

  Example of use:

  - */IndoorLocation/getMapPNG?building=Alumni_Hall&floor=AM-00*

5.5.2. Generic APIs (database entities)

In addition to the APIs for the different sections of the system, which are described in previous and subsequent sections of this document, the BOSSA Platform offers a series of APIs which are linked to the main database schema for the Indoor Location System (*indoor_location_db*). In this document, they are called *generic APIs*:

- Beacon_atmosphere
- Beacon_info
- Building_info
- Device_location
- Device_section
- Device_voltage
- Sergent_battery
- Sergent_event
- Sergent_info
- Sergent_status

Like any other API, the generic APIs are made of a model and a controller. However, for them the model is linked to a certain entity in the database. Each one of the generic APIs created in the Platform is directly linked to a table in the main database schema and has its same table name and fields.
Sails allows to automatically link to a database the APIs generated in the way described. In order to do that, it is enough to modify the file *connections.js*, in the *config* folder. These configurations try to promote the use of the Model-View-Controller (MVC) structure.
As described in previous sections of this document, most of the BOSSA_Platform application's logic is condensed in APIs which are not directly linked to database entities (*non-generic APIs*). This was a development and design decision. Due to the nature of the project and the requirements of the platform, it was considered

that the development methods and structure selected were the most desirable. Furthermore, some of the APIs developed required access to different database schemas, therefore generic APIs did not seem to be the optimal approach.
However, the basic generic APIs were created to be offered for future BOSSA developers. If they considered that the use of generic APIs provides benefits to the APIs' structure or to the Platform operations, they are welcome to make use of the available generic APIs and to create new ones that imply more complexity.

5.5.3. Caller App (or User App)

The Caller App (or User App) is used to perform emergency calls to 911, informing of the indoor location of the caller.
When a user places an emergency call using the Caller App, the smartphone application first scans for beacons. Then it sends to the Location Server, which is included in the BOSSA Platform, the identifications of all the beacons that it sees, together with their Received Signal Strength Indicators (RSSI). The RSSI shows the strength of a Bluetooth signal received. The data is sent using an API provided by the BOSSA Platform. The Platform interacts with the main database (*indoor_location_db*) to learn the location of each BLE device in the set and applies an algorithm to the locations and RSSIs to determine either the location of the closest beacon or the actual x, y coordinates of the caller.
Once the address and the indoor location of the caller are identified by the location server module and returned to the smartphone application in PIDF-LO XML format (Figure 20), the phone initiates a call to a test-bed in the IIT Real-Time Communications Lab (RTC Lab) that replicates the functions of an ESInet.
The call is IP-based and uses the Session Initiation Protocol (SIP) [17] for its signaling component, and the Real-time Transport Protocol (RTP) for its audio and video component.
When the PSAP operator answers the call, the civic address as well as the room, floor and room number of the caller are displayed on the PSAP operator's computer screen.
The civic address is used to route the call to the correct PSAP in accordance with the i3 standards.
The mobile application is built on the Android OS and includes the functions of a SIP User Agent, which are provided by the open source *SIPdroid* application. [1]

```
<presence
    xmlns="urn:ietf:params:xml:ns:pidf"
    xmlns:gp="urn:ietf:params:xml:ns:pidf:geopriv10"
    xmlns:ca="urn:ietf:params:xml:ns:pidf:geopriv10:civicAddr"
    xmlns:gml="http://www.opengis.net/gml"
    entity="sip:caller@64.131.109.27">
    <tuple
        id="id82848">
        <status>
            <gp:geopriv>
                <gp:location-info>
                    <ca:civicAddress>
                        <ca:country>
                            us
                            </ca:country>
                        <ca:A1>
                            IL
                            </ca:A1>
                        <ca:A2>
                            Chicago
                            </ca:A2>
                        <ca:A6>
                            35th
                            </ca:A6>
                        <ca:PRD>
                            W
                            </ca:PRD>
                        <ca:STS>
                            St
                            </ca:STS>
                        <ca:HNO>
                            10
                            </ca:HNO>
                        <ca:LOC>
                            RT-9-9C3-1-0603 Last Seen: 0 seconds ago
                            </ca:LOC>
                        <ca:FLR>
                            9
                            </ca:FLR>
                        <ca:ROOM>
                            9C3
                            </ca:ROOM>
                        </ca:civicAddress>
                    </gp:location-info>
                <gp:usage-rules/>
                <gp:method>
                    Manual
                    </gp:method>
                </gp:geopriv>
            </status>
        <contact
            priority="0.8">
            sip:caller@64.131.109.27
            </contact>
        <timestamp>
            2016-11-07T23:22:23.500Z
            </timestamp>
        </tuple>
    </presence>
```

Figure 20 – PIDF-LO XML showing the caller's location.

The Main Screen and the Call Screen of the Caller App are shown in Figure 21. Three buttons on the Main Screen allow the user to place a call, to see the status of the call and to add more information via a text interface if desired.

When the user selects the "911 Call!" button, the smartphone's operating system scans for BLE beacons and the described process starts. The beacon identifiers and RSSIs are sent used a GET HTTP request to the correspondent API, and the formatted XML received as answer is inserted in the body of the SIP message (INVITE message) that is used to place the call to the ESInet. [1]

For more details about the internal processes performed by the Caller App, see the document in the reference [2].

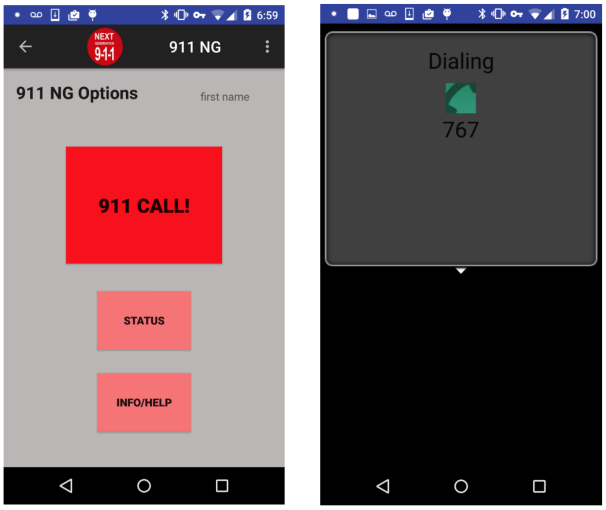

Figure 21 - Caller (User) App. Main screen (left) and Call screen (right)

5.5.4. Base technology transition

The base application for the Caller App is the open-source code Sipdroid, as described in previous sections of this document.
However, in recent projects, some deficiencies of the base technology have been observed. The main issue implies incompatibility with certain codecs. Furthermore, it is intended to include the functionality of calculation and submission of the indoor location also during the call and not only during the call establishment. For this purpose, SIP Re-Invites will be used as previous step, and the SIP INFO Method as final solution. For more information about the reasons for the use of the mentioned SIP functions, see the document under the reference [3].
Including the described functionalities in the current application implies high complexity, due to the fact that it is based on code used for previous projects and it is not specifically designed for the current system. The application also shows a certain degree of software instability.
For this reason, and considering the observed incompatibilities, it has been studied the potential benefits of the redesign and reconstruction of the application, taking a different application or SIP User Agent emulator as base technology. For more information about this study, see the document under the reference [5].

5.5.5. WebRTC Caller App

The WebRTC Caller App is a web-based smartphone application that makes a WebRTC [18] connection to a WebRTC-based PSAP that needs to obtain the caller's indoor location.

The application is aimed to be a WebRTC version of the Android Caller Application for emergency calls to a WebRTC server. Therefore, its execution follows an equivalent procedure.
When a user places an emergency call using the WebRTC Caller App, it first scans for beacons. Then it sends to the Location Server module, which is included in the BOSSA Platform, the identifications of all the beacons that it sees, together with their Received Signal Strength Indicators (RSSI). The data is sent using an API provided by the BOSSA Platform. The Platform then interacts with the main database (*indoor_location_db*) to learn the location of each BLE device in the set and applies an algorithm to determine either the location of the closest beacon, the actual x,y coordinates of the caller, or both of them.
Once the address and the Indoor Location of the caller are identified by the location server module and returned to the smartphone application in JSON format, the phone initiates a WebRTC connection to the WebRTC Emergency Service, indicating the location of the caller. The location is displayed on the screen of the WebRTC PSAP.
For more information about the WebRTC Caller App and WebRTC Emergency Service System, see the reference [6].

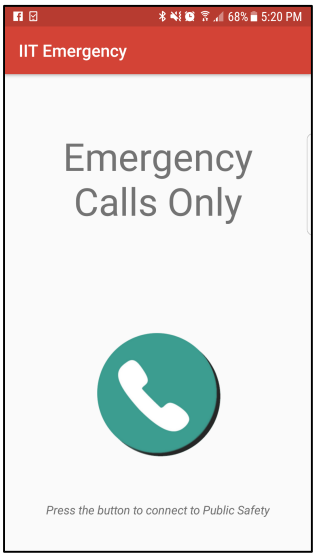

Figure 22 - WebRTC Caller App. Main screen.

5.5.6. IndoorLocation Algorithms

Four algorithms for determining the indoor location of a caller were implemented and tested. The first three identify the location of the iBeacon closest to the caller. The fourth identifies the location of the caller using x,y coordinates and a predetermined building floor origin.

The first step in the process of indoor location estimation is identifying the correct floor. For this purpose, the *power sum* and *flr cnt* methods are used.
The second step is identifying the particular location on the floor. Each of the four algorithms takes a different approach for this step [1]:

- The Maximum RSSI algorithm, **MAX RSSI**, selects the BLE device with the highest RSSI during a 10-second sample period. The algorithm notes the identity of the device associated with that value and queries the database for the description of the indoor location associated with that identity. This algorithm is also referred to as minRSSI.
- The Average Power in dB algorithm, **AVG DB**, selects the device with the highest average power as measured in dB over the sample period.
- The third algorithm, Average Power in dB algorithm, **AVG MW**, selects the BLE device with the highest average RSSI as measured in milli-watts. The values of the RSSI's are first converted to milli-watts, then the milli-watt values are averaged, the maximum is found, and the associated BLE device is chosen to be the one closest to the caller.
- The fourth algorithm, Trilateration with Least Squares fitting, **TRI LS**, attempts to locate the caller rather than the beacon closest to the caller. The location is given as a floor number together with the caller's x,y coordinates on that floor relative to a predefined origin. The x,y coordinates of each BLE device are recorded in the Location database along with the description of the room and floor on which the device is deployed. After the floor is identified, a process of trilateration and least squares fitting is used to calculate the estimated x,y coordinates of the caller.

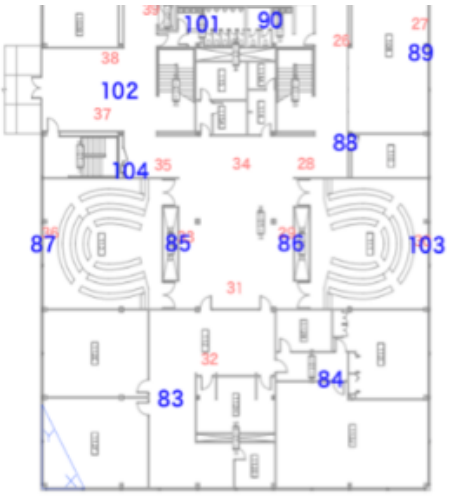

Figure 23 - Floor map prepared for algorithms' test. Beacons' locations in blue and test locations in red.

Today (July 2017), the BOSSA Platform offers APIs implementing the Maximum RSSI algorithm (*minRSSI*) and the Trilateration with Least Squares fitting algorithm (*LeastSquared*).
For more information about the algorithms, as well as about the tests performed and the conclusions from their results, see the document from reference [1].

5.6. Indoor Location testing and analysis

This section covers the test and analysis tools available for the Indoor Location System.
The platform's partial diagram in Figure 24 shows the systems involved in the section. It includes the two main database schemas, the Test App and the APIs with which it interacts with the BOSSA Platform.
The subsequent subsections describe the APIs for testing and data analysis purposes implemented in the Platform, as well as the Test App main goal and operations.

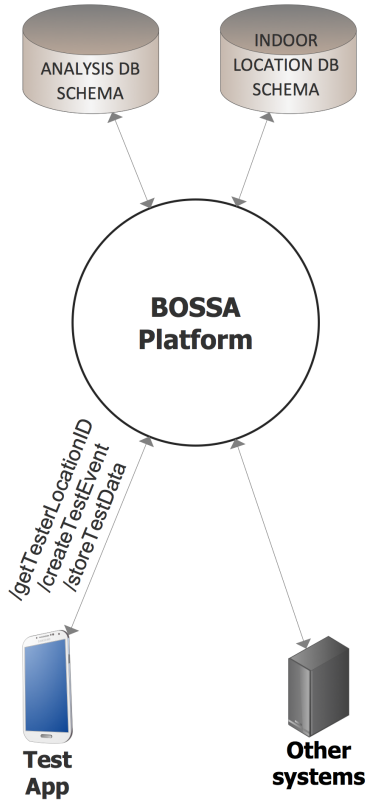

Figure 24 - Indoor Location testing and analysis. Partial platform's diagram.

5.6.1. IndoorLocationTest APIs

The following APIs are implemented in the IndoorLocationTest controller, and give service to the testing and analysis tools for the Indoor Location System:

- **/getTesterLocationID**?building_acr=[building_acr]

  Allows to get the correspondent Tester Location ID from the database schema *testing_application_db*. It returns basic building information.
  It makes use of a view designed in the database for this specific purpose (*'view_locId_buAcr'*).
  Then the results are filtered in order to show only the data from the building specified as parameter (building_acr).
  One input parameter is mandatory when calling the API: building_acr. It corresponds to the building acronym, stored as *'tester_build_acr'* in the table *'experiment_tester'*.

- **/createTestEvent**?runnumber=[runnumber]&testlocationid=[testlocationid]

  Performs changes in the analysis database *testing_application_db*.
  Creates a new record in the table '*experiment_event'*, including the parameters in the columns *'tev_fk_run_id'* and *'tev_fk_tester_id'*, respectively.
  Both input parameters are mandatory in the request: runnumber and testlocationid. They are the experiment run id and the experiment tester id, respectively.

- **/storeTestData**?jsonArray=[{major:[major1], minor:[minor1], rssi:[rssi1], testnum: [testnum1]} , {major:[major2], minor:[minor2], rssi:[rssi2], testnum: [testnum2]} , …]

  Stores the data resulting from the test experiments performed by the Test App, in the database *testing_application_db*, in the table *'experiment_datapoint'*.
  The input parameter jsonArray is mandatory in the request. It should include an array of JSONs, each one of them specifying the major, minor, rssi and testnum of an iBeacon.

5.6.2. Test App

The purpose of the Test App is to allow a tester to generate data sets in different locations of a building. The data is collected and stored in the experiment database schema (*testing_application_db*). All the data stored from test sessions can be used for analysis purposes, failures' detection and location algorithms' improvements.

The algorithms' testing process starts with the selection of the positions in which the tests will be done. Testers stand at each test position and trigger a scan for Bluetooth beacons, in the same way that the Caller App does (User App).
The resulting set of data is sent to an experiment database schema, which currently (July 2017) is *testing_application_db*.
Then, the algorithm developers can use the datasets in the database as input to different algorithms, which are explained in previous sections of this document. Since the exact location of the test position is recorded, it can be compared to the results of the algorithm's estimation, and the accuracy of the algorithm can be put in numbers. The measure of accuracy is taken to be the distance between the actual position (known before hand) and the position estimated by the algorithm.
For more information about the Test App, see the reference [7].

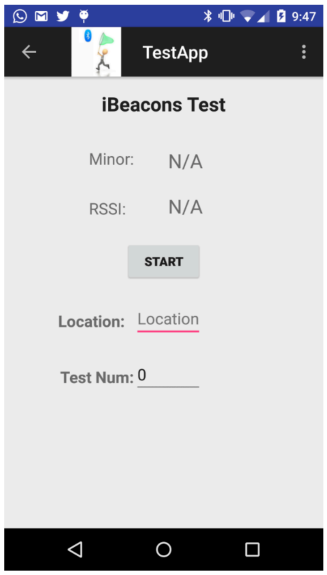

Figure 25 - Test App. Main screen.

5.7. Atmosphere data

This section deals with the extraction, storage and retrieval of atmosphere data, which includes temperature and humidity information. The atmosphere data comes from the sensors integrated in the second-generation iBeacons.
The diagram in Figure 26 shows a representation of the elements involved in this section: the main database schema for the Indoor Location System, the Environmental App, including the APIs used by it, and other systems that make use of the atmosphere resources.
In the following subsections the APIs for Atmosphere available in the BOSSA Platform are described, as well as the Environmental App, which makes use of the APIs. Furthermore, the process for Atmosphere data retrieval and the devices' structure configurations are discussed.

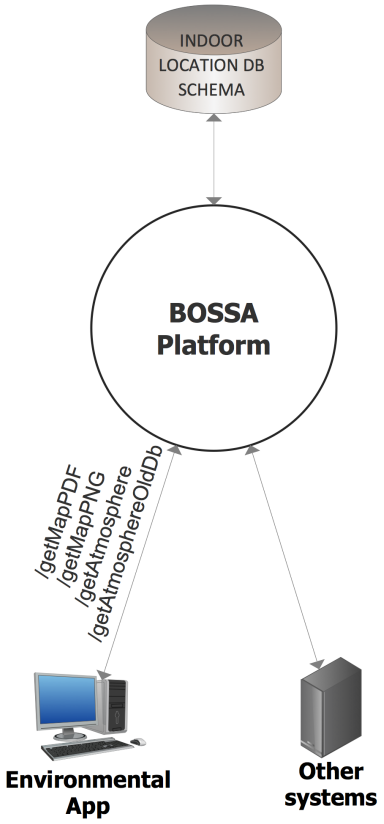

Figure 26 - Atmosphere Data. Partial platform's diagram.

5.7.1. Atmosphere APIs

The Atmosphere APIs' logic is implemented in the Atmosphere controller, and allow the atmosphere data retrieval, specifying several different conditions.

- **/getAtmosphere**
  or
  /getAtmosphere?values=[values]
  or
  /getAtmosphere?buiding=[building]&floor=[floor]
  or
  /getAtmosphere?buiding=[building]&floor=[floor]&values=[values]

  If no parameters are included in the request, the atmosphere (temperature and humidity) data and other relevant fields are retrieved, from all the entries

in the table *'beacon_atmosphere'* from the main database schema *indoor_location_db*.
The information is retrieved by executing the database view named *'view_atm_data'*, which was specifically designed for this purpose (Note for future BOSSA developers: if any change is performed in the database, it is strongly recommended to also perform the correspondent changes in the views involved in the db changes).
In the case that building and floor parameters are included in the request (then both are needed), the data is filtered by building (*'bu_name'* in table *'building_info'*) and floor (*'loc_floor'* in table *'device_location'*). The building should be specified by name and the floor by number.

Example of use:

- /atmosphere/getAtmosphere?building=Alumni_Memorial&floor=2

If the parameter values is set in the request to the value last, only the last value registered for each iBeacon is returned. In any other case, all the data registered for the iBeacons is returned.

Examples of use:

- /atmosphere/getAtmosphere?values=last
- /atmosphere/getAtmosphere?building=Alumni_Memorial&floor=2&values=last

- **/getAtmosphereOldDb**

  It provides atmosphere (temperature and humidity) data and other relevant information from all the available iBeacons in the table *'device_atmosphere'* from the transitional database *indoor_location*.
  It executes a view (*'view_atm_data'*) in the database schema designed to provide the desired data.
  This API is temporal and it is used while transitioning between database schemas. Its deletion is recommended, once the new database schemas *indoor_location_db* and *testing_application_db* are consolidated and working for every functionality of the Indoor Location system.

5.7.2. Environmental App

The Environmental App offers an interface to display temperature and humidity data from a building, as well as the correspondent maps.
The application is built using *Qt* framework and *C++* programming language.
To learn more about the purpose and development of the Environmental App, see the document in reference [4].

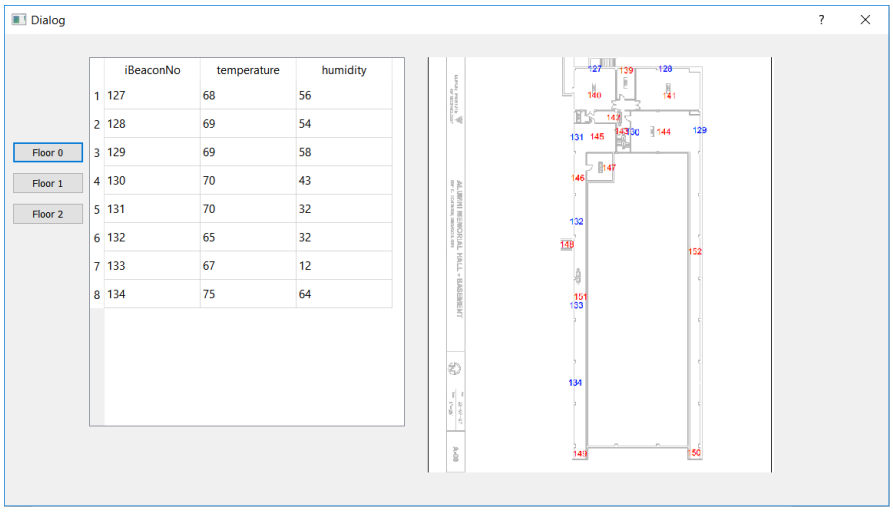

Figure 27 - Example screen of the Environmental App.

5.7.3. Atmosphere data generation and storage

As it is described in previous sections of the document, the scope of the Indoor Location Project increased with the second-generation iBeacons, which incorporate sensors for temperature and humidity.
The built infrastructure allows to use the iBeacons' array as a sensors' array to offer useful atmosphere data from the buildings in which the beacons are deployed. This use is added to the system, as a new conceptually different function than the indoor location service.
The first-generation beacons where deployed following a flat structure. Each of them performed the same function of Bluetooth advertisement, sending every half second its identification (major, minor and uuid).

However, for the second-generation beacons, a hierarchic structure was designed. The new structure defines two types of devices:

- **Second-generation iBeacons**, which perform the Bluetooth advertisement function and take temperature and humidity samples regularly.
- Gateway devices or **Sergeants**. Each of the sergeants has a group of iBeacons associated (section) and it "collects" their atmosphere data regularly. For that purpose, it creates a temporal direct Bluetooth connection with each of the beacons. Once the section's data is gathered, the sergeant makes use of the Wifi network to send it to the Particle cloud service, where the data is stored.

In order to retrieve the atmosphere data from the Particle cloud, the cloud tool Webhook is used. It is configured with one of the APIs available in the BOSSA

Platform. In this way, the atmosphere data is sent automatically to the platform, which processes it and stores it in the correspondent database schema (*indoor_location_db*).
The image in Figure 28 shows an example of a building's floor with beacons and sergeants deployed. It can be observed how each of the sergeants (black) is in charge of a group of beacons (blue). The sergeants' distribution in the map, the number of sergeants needed for a floor, as well as the number of iBeacons per section is a topic under research. After the first experiments it has been observed that these parameters need to be studied for each case, taking into consideration, between others, the structure of each floor.
To know more about the devices array structure and the experiments performed, see the document in reference [4].

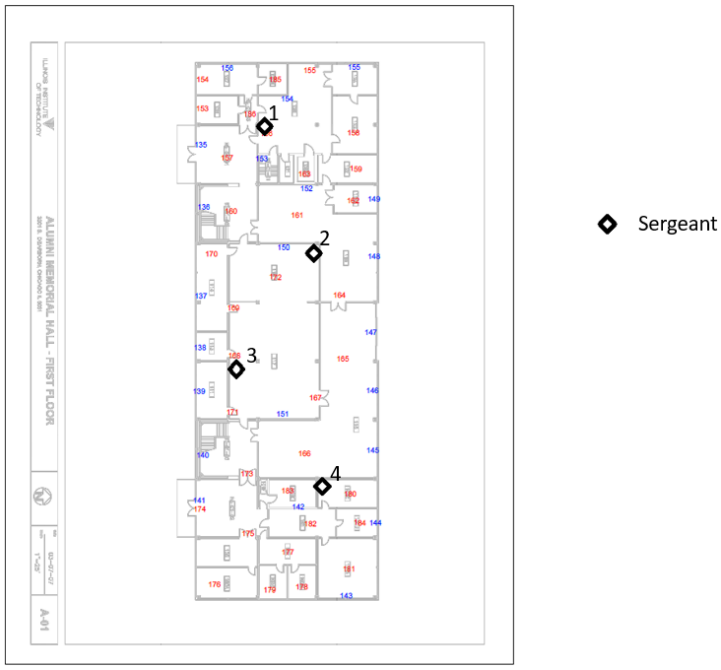

Figure 28 - Floor map showing sergeants (black), beacons (blue) and test locations (red).

5.8. Deployment and Management

This section of the system covers the deployment and management of devices in buildings, performing updates in the databases.
The diagram in Figure 29 shows the elements involved with the section: the Indoor Location (*indoor_location_db*) and Authentication database schemas, the Indoor Location Management Server, the Deployment App and the Particle could services using the Webhook tool.

In the following subsections the Management APIs in the BOSSA Platform are described. Also the Deployment App and the Management Web-App are depicted, which make use of the Management APIs.

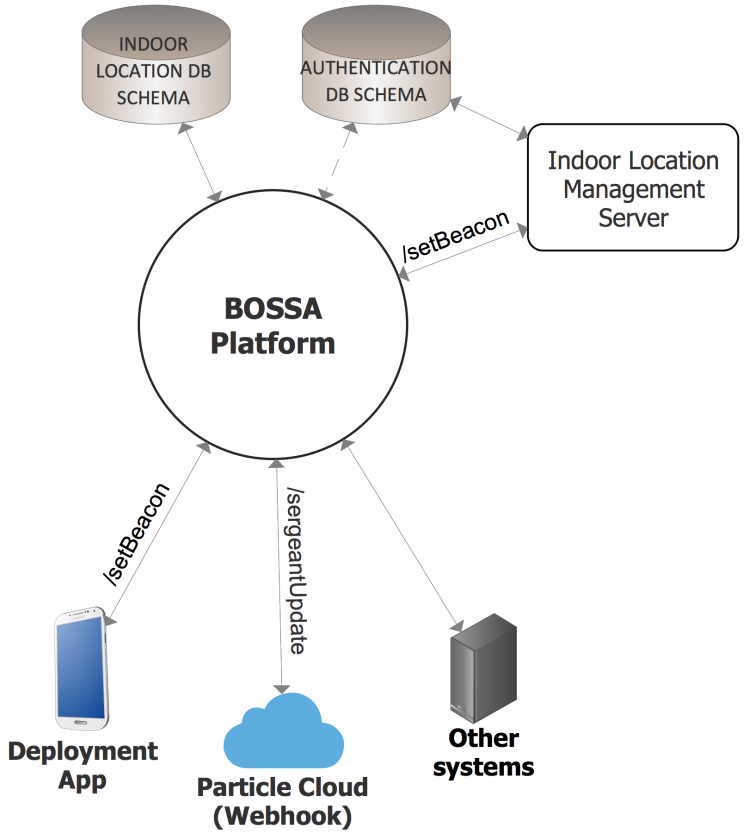

Figure 29 – Deployment and Management. Partial platform's diagram.

5.8.1. Management APIs

The following APIs' logic is implemented in the Management controller and they deal with management and deployment of devices in buildings.

- **/setBeacon**?major=[major]&minor=[minor]&locId=[locId]

  Allows to change the location of a beacon to a certain location. The location of the beacon previously deployed in that location is set to null, which implies it is not deployed anymore.
  All the input parameters are mandatory in the request: major and minor of the beacon, and identifier of its new location (locId).

Example of use:

- /management/setBeacon?major=2000&minor=577&locId=100

- **/sergeantUpdate**?json=[json]

  Allows to send atmosphere (temperature and humidity) data of a certain beacon, so it can be stored in the correspondent tables in the Indoor Location database schema (*indoor_location_db).*
  This API is specifically designed for its use by the Particle cloud service's tool Webhook, which uses the API regularly to send the information received from the sergeants.
  The format for the json parameter is particular to the needs of the could service. It includes values for the MAC address of the beacon (mac), the temperature and humidity values (temp and humidity), as well as the voltage and battery of the device (voltage and battery), if reported.

5.8.2. Deployment app

The Deployment App was design in order to ease the iBeacons' deployment process and to minimize the possibility of human error when updating data in the database.
The deployment procedure consists on the identification of each iBeacon to deploy, identification of the corresponding locations for each of them, and placement of the iBeacons in their assigned locations in the building.

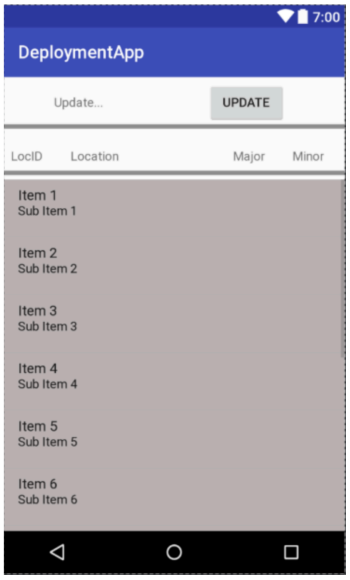

Figure 30 - Deployment App main screen

The app allows the deployer to store the deployment data in the correct database in order to keep track of the iBeacons' locations.
Currently (July 2017), the database schema in which the data is stored is *indoor_location_db*.

The Deployment App is implemented in the Android OS and its main screen shown in the Figure 30.

5.8.3. Management Web-App

The Management Web-App is in the Indoor Location Management Server and consists in a web application created for the deployment and management of devices in buildings.
For this purpose, it offers a form, with which data modifications can be requested. This form makes use of APIs from the BOSSA Platform, which makes the changes in the correspondent database.
The BOSSA Platform shows a link in its Front-end in order to access the Management Web-App, which requires user authentication.
The Management Web-App is a convenient tool for devices' deployment. Its web nature makes it multiplatform, which is a differentiator from the Deployment App, developed in the Android OS.

5.9. Frontend

This section covers the Frontend, which can be found under the subdomain *api.iitrtclab.com*. It is designed to publish the available resources on the BOSSA Platform and to offer documentation about the project. The webpage is also designed to offer a tool for management and deployment of devices in buildings for the Indoor Location System.
The Front-end makes internal use of APIs in the platform for operations centralization purposes. This fact is illustrated in the diagram in Figure 31.
Integrating the Frontend in the application by using APIs internally was a design decision. This decision was based on the fact that the Front-end is mainly informative and the requirements do not include heavy processing or busy traffic expectations.
If in the future the requirements for the Front-end changed, it is suggested to future BOSSA developers to move the frontend part of the code to a different server, and put it under the general domain *www.iitrtclab.com*.
In the following subsections, the main parts of the main webpage are described. Also the internal Frontend APIs are presented.

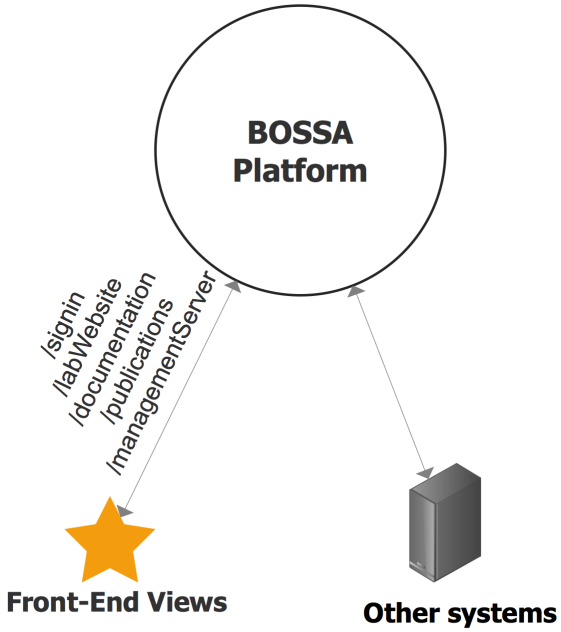

Figure 31 - Frontend. Partial platform's diagram.

5.9.1. Main webpage

The main webpage for the BOSSA Platform is shown in Figure 32. It shows a welcome message and a link to the Real-Time Communications Lab webpage. Just below, the site has the following sections:

- **Platform Documentation**: contains the platform's APIs documentation.
- **Projects & Publications**: pool of relevant documents concerning the projects in the RTC-Lab, including the BOSSA Platform project.
- **Indoor Location Management**: link to the Indoor Location Management Server, which allows to perform deployment and management operations for the Indoor Location System. It requires user authentication.

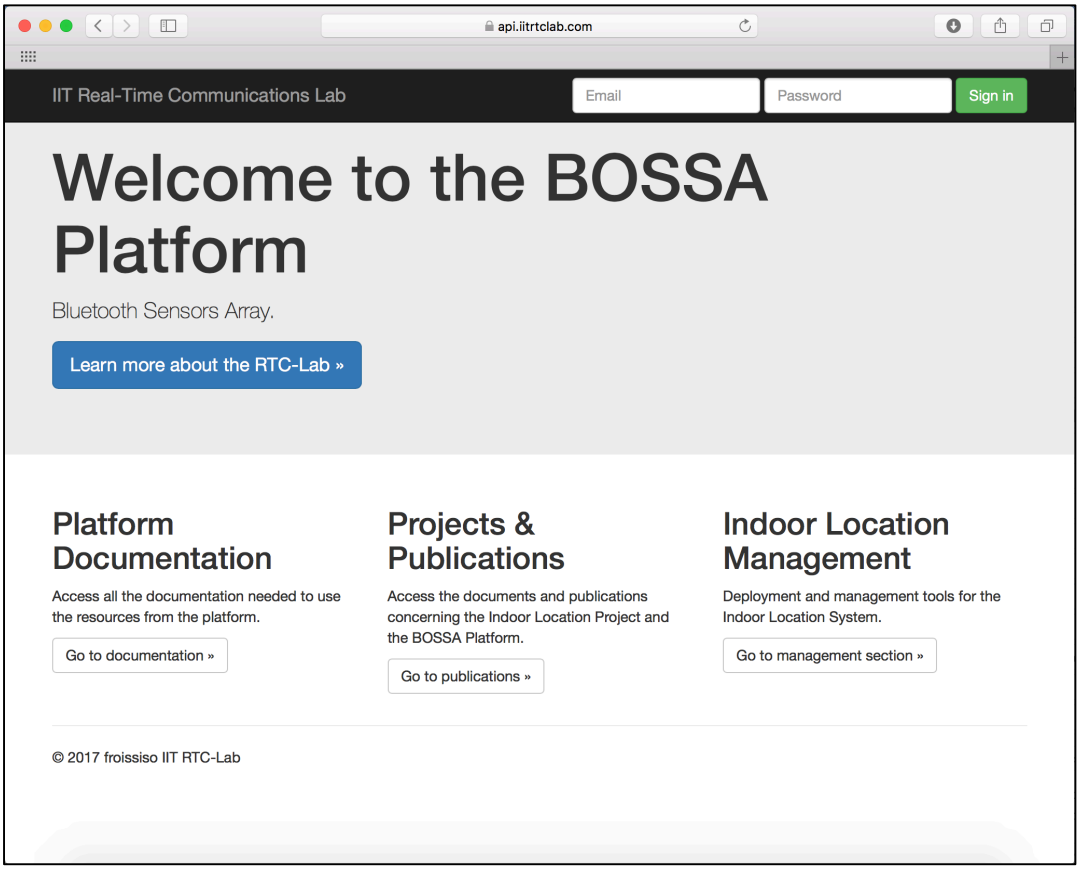

Figure 32 - BOSSA Platform Front-End. Main screen.

5.9.2. Frontend APIs

The following APIs' logic is implemented in the Frontend controller. Their purpose is to serve internally the Front-end of the platform, using a centralized approach:

- **/signin**
- **/labWebsite**
- **/documentation**
- **/publications**
- **/managementServer**

6. Platform Integration and Deployment

The purpose of this section is to explain the main points of the deployment procedure for the platform to the cloud services. Moreover, the domain name used is shown and it is suggested to future BOSSA developers a procedure for creation and publication of new systems. These objectives are explained in subsequent subsections.

6.1. AWS instances

Today (July 2017) there are two AWS EC2 instances running and published:

- **BOSSA Platform**: contains the code for the BOSSA Platform, cloned (or pulled) from the correspondent git repository: *https://github.com/IIT-RTC-Lab/BOSSA_Platform*
- **Indoor Location Management Server:** contains the code for the server for management and deployment of devices of the Indoor Location System. A copy of the code can be found on the git repository: *https://github.com/IIT-RTC-Lab/Indoor_Location_Management_Server*

Each of the instances has a *keypair* file associated, needed for access. Each *keypair* can be found on the correspondent instance's repository.
Both instances run the Amazon Linux distribution. Node.js, Sails.js and the modules and libraries needed were on the instances.
Furthermore, both instances currently have a Nginx proxy installed and configured. The proxy receives requests on the default HTTP port 80. The proxies are configured to only allow secure HTTPS requests. Nginx provides a security certificate using Certbot. Hence, uncertified HTTP requests will not be allowed to reach the server.
The accepted requests are routed by the proxy to the port 1337, in which the server is listening and in continuous execution.
As a note for future BOSSA developers, it is important to keep in mind that if for any reason it was needed to reboot any of the running instances, it would be also needed to start again the Nginx proxy. It should be enough with the execution of the command: *$ sudo service nginx start*

On the other hand, a WebRTC server for Emergency Services is currently under development. Today it is behind the domain name *911webrtc.com.* In the future a new EC2 instance will be created for this module, as well as a subdomain with a meaningful name. This is explained in the following section of the document.

As a note for future developers of the Indoor Location Project, it is suggested to follow the following steps in order to deploy new code to the AWS EC2 instances already created and running:

- Connect to the instance using ssh commands and the correspondent *keypair* file.
- Stop the running server (($ forever stopall OR $ forever stop app.js).
- Synchronize server with correspondent git repository ($ git pull).

For the continuous execution of the deployed applications, the utility forever was used. In order to install the utility in a new machine it is enough with executing the following command: $ npm install forever –g . Most commonly used commands are described in the following:

- $ forever start app.js : initiates continuous execution of the node application.
- $ forever stop app.js: stops application running continuously from app.js.
- $ forever stopall: stops all forever processes.
- $ forever list: shows a list of all forever processes running.
- To learn more about the available commands, execute $ forever –help

During the BOSSA Platform development process the Heroku cloud services were used. They are a useful tool during development due to the fact that the configurations for automatic code deployment linked to push actions to the git repository, are extremely simple to do. Using Heroku services can be a good option for future BOSSA developers as a previous step before the final deploy of the services to the AWS instance and the correspondent subdomains under iitrtclab.com .

As a final note on this section, it is important to mention that the BOSSA Platform can be deployed in two different modes: *development* and *production*.
Conceptually, *production* would be the correct mode for deployment. However, considering that the project is in constant development and experimentation process, it has been decided to deploy the platform in *deployment* mode.
Despite keeping the platform in *development* mode, the *blueprints* have been disabled for security reasons. As it is explained in previous sections of the document, *Sails blueprints* are automatically offered to Sails developers to perform basic actions in the APIs' controllers (CRUD methods): find, create, update and destroy. These actions are useful during development, but once the server is deployed they can imply a security threat.
In the case that future developers decided to deploy in *production* mode, it is suggested that they follow the instructions detailed in reference [34].
In any case, it is always important to keep in mind that the Nginx proxy is configured to receive HTTPS requests in the default port 80, and to forward them to the port 1337, in which the application should be listening. Therefore, it is recommended to keep the listening port in the application configurations to 1337 and avoid the point from the instructions referenced before, in which it is indicated to change this configuration to port 80.

6.2. Domain and subdomains

The domain name *iitrtclab.com* is the one used for publishing of the BOSSA Platform and also other systems of the Indoor Location Project.
Each module of the system has been placed behind a subdomain with a name that represents its function:

- BOSSA Platform (APIs): *api.iitrtclab.com*
- Indoor Location Management Server: *management.iitrtclab.com*

In the near future, as discussed in the previous section, the module WebRTC Server for emergency services will be installed in a new EC2 instance and will be placed behind a proper subdomain.
As a note for future developers of the Indoor Location Project, it is suggested to follow the procedure described for the creation and publication of new modules or systems:

- Create an Amazon EC2 instance.
- Deploy the server or application developed in the instance created and set it to run continuously.
- Create an Elastic IP for the instance.
- Access AWS Route 53 and link the elastic IP of the instance with a meaningful subdomain name.

7. Database

The main database used by the BOSSA Platform and Indoor Location System is stored in an Amazon Web Services (AWS) RDS Instance.
It contains several database schemas for different purposes. They are explained in the following subsections, as well as the database views employed and the strategy suggested for database's queries generation.

7.1. Database schemas

The system has database schemas to store information from the Indoor Location System, data from the system that could be used for analysis purposes, as well as users' authentication data.
With the platform's creation and the system's scope expansion, it was decided to redesign the database. Two conceptually independent database schemas were conceived:

- **Indoor Location DB Schema** (*indoor_location_db*): storage of information about the system and its elements. See diagram in Figure 33.
- **Indoor Location Analysis DB Schema** (*testing_application_db*): for storage of experimental data, which can be used for analysis purposes and for indoor location algorithms' improvement. See diagram in Figure 34.

In addition to the redesign and division of the previously existent schema, new tables were introduced in order to give support to the new devices for the Indoor Location System (sergeants and AXA beacons) and to store atmosphere data (temperature and humidity) from the second-generation iBeacons.
Moreover, names for tables and fields were changed in order to comply with the *good practices* for relational databases.
To know more about the database and the *good practices* applied, see the document in the reference [8].
Apart from the two DB schemas described, there is a third: **Authentication DB Schema**. Its objective is to store user authentication information for the Indoor Location Management Server, and in the future for every function in the system that required access authentication for users.

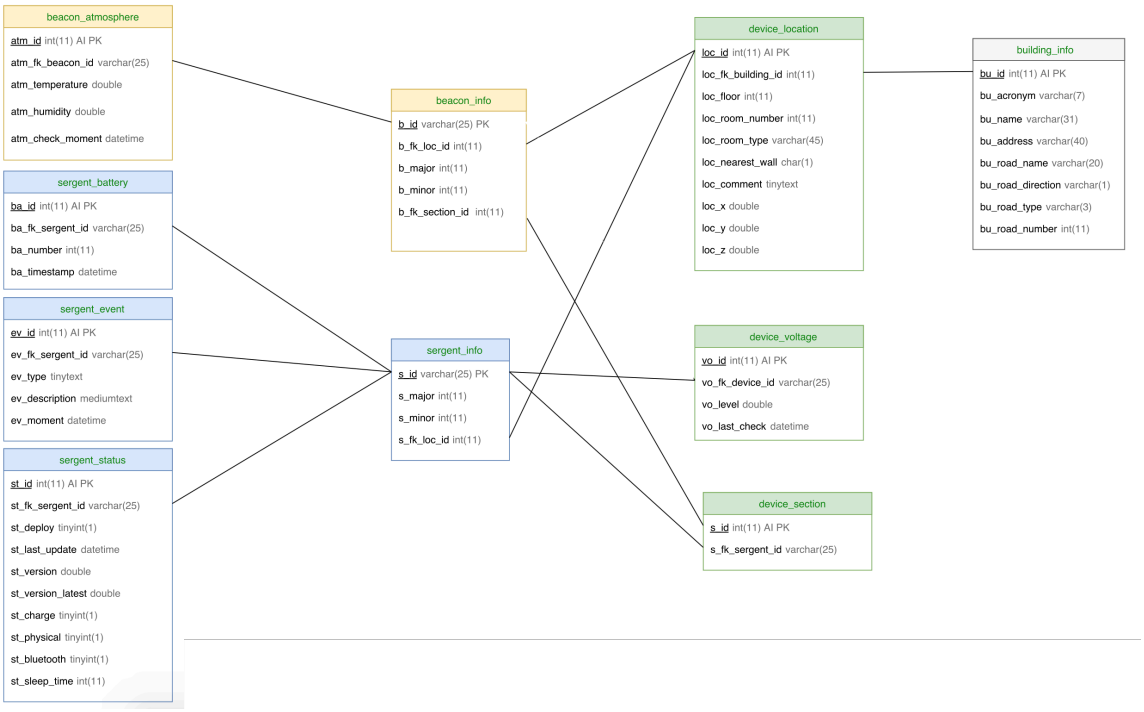

Figure 33 - Diagram for the Indoor Location DB Schema

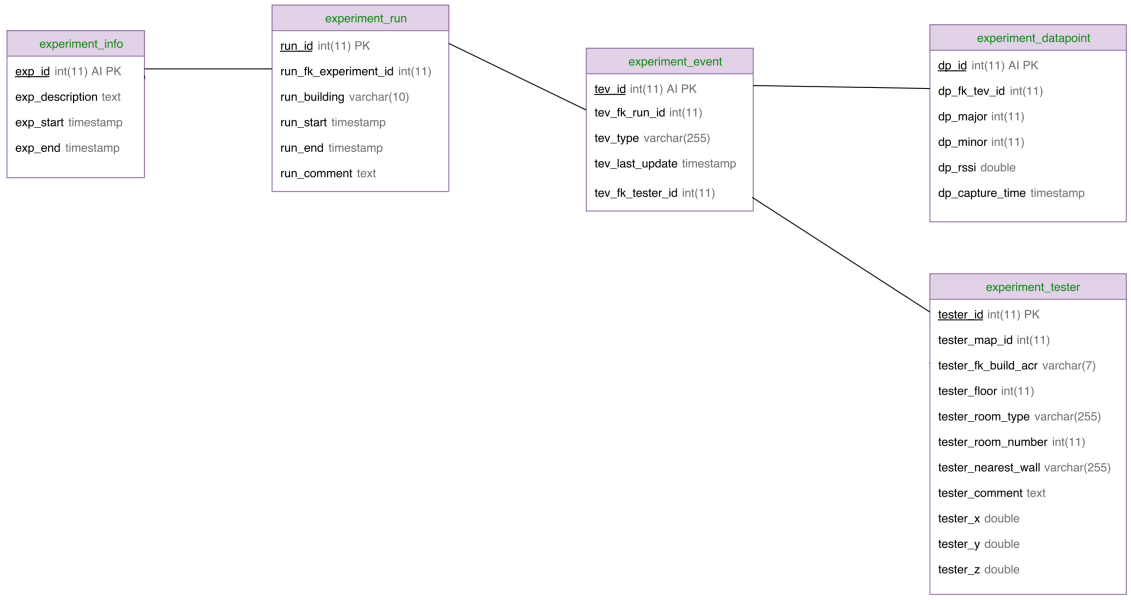

Figure 34 - Diagram for the Indoor Location Analysis DB Schema

7.2. Database Views

In order to achieve modularity and decoupling between the different parts of the Indoor Location system, it was made the decision to design the database queries as database views. Database views are scripts, created in the database they refer to.

In this way, when a developer needs an API to execute a complex query to the database, he/she will implement the logic in the database side and then call the DB from the API's code. Then the query in the DB view will be executed.

The purpose of this approach is to isolate the code from the databases as much as possible. When changes are performed in the database, the views that use the tables involved would be updated directly in the database side, by the same database designers that originally performed the modifications. Therefore, the applications that execute the DB views will not perceive any change and will not need any code modification. In the practice, the code will execute the query to the DB view as it was a "black box", which returns the result, independently of how it was reached.

As an example taken from the Indoor Location Platform, the API `/getAtmosphere` from the IndoorLocation APIs section, has the purpose of retrieving specific data from a table, as well as data from other tables related to the entities from the first one. This is achieved by doing several "joins" In this case, instead of implementing the query directly in the API's code, which would imply access to several fields in several different tables, all the logic is implemented in the view called *'view_atm_data'.* In Figure 35 it is shown the extract from the platform's code in which the view is called, and the implementation of the view.

This example shows the decoupling reached between platform code and database. In the case that any of the tables is altered or the atmosphere data is required to be accessed in a different way, the only code to change would be the DB view's. The sql query in the platform's code would remain the same, independently of the database alterations.
Other example of view implemented in the 'indoor_location_db' is *'locID_buAccr'* which retrieves location ids and the associated building acronyms.

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = `clement`@`%`
    SQL SECURITY DEFINER
VIEW `indoor_location_db`.`view_atm_data` AS
    SELECT
        `T1`.`atm_id` AS `atm_id`,
        `T2`.`b_id` AS `b_id`,
        `T1`.`atm_temperature` AS `atm_temperature`,
        `T1`.`atm_humidity` AS `atm_humidity`,
        `T1`.`atm_check_moment` AS `atm_check_moment`,
        `T2`.`b_major` AS `b_major`,
        `T2`.`b_minor` AS `b_minor`,
        `T3`.`loc_id` AS `loc_id`,
        `T3`.`loc_floor` AS `loc_floor`,
        `T4`.`bu_name` AS `bu_name`,
        `T4`.`bu_address` AS `bu_address`
    FROM
        (((`indoor_location_db`.`beacon_atmosphere` `T1`
        JOIN `indoor_location_db`.`beacon_info` `T2` ON ((`T1`.`atm_fk_beacon_id` = `T2`.`b_id`)))
        JOIN `indoor_location_db`.`device_location` `T3` ON ((`T2`.`b_fk_loc_id` = `T3`.`loc_id`)))
        JOIN `indoor_location_db`.`building_info` `T4` ON ((`T3`.`loc_fk_building_id` = `T4`.`bu_id`)))
```

```
            if(Object.keys(parameters).length === 0){
                console.log('No parameters in the request');

                //execute view from mysql db to show relevant atmosphere data
                con.query('SELECT * FROM indoor_location.view_atm_data;',function(err,rows,fields){
                    if(err) throw err;

                    console.log('Data received from db\n');
```

Figure 35 - Example of database use. DB view implementation (above). DB view call from an API code (below).

8. Conclusions and future work

The platform developed in this project, BOSSA Platform, works and it is integrated with the other elements of the Indoor Location System. Information about how to make use of the resources of the platform is accessible and public on a website (*api.iitrtclab.com*).

BOSSA allows external developers to access the resources of the Indoor Location Project through the use of APIs, and the needed documentation for that purpose is clear and easily reachable.
The platform performs the Location Server's role for the Indoor Location System and provides support to maintenance and test services.
The system has been adapted to accommodate the new devices and their architectures for temperature and humidity data acquisition, and BOSSA includes functions for collection, storage and retrieval of the data obtained.
In addition, processes and internal interactions have been standardized and modularity of the elements of the system has been fostered, making the components to interact through the platform. Moreover, the technologies used to build the platform allow security configurations in order to avoid undesired uses or intrusions.
Finally, the platform improves work flow efficiency as it is implemented in a single programming language (JavaScript), and it includes Backend and Frontend in the same web application.

The BOSSA Platform fulfills the requirements defined. Thus, the project's goals have been accomplished.
On the one hand, the project is now "open" for external developers in a controlled manner. On the other hand, all the work developed has been explained and working procedures have been described in order to allow internal developers of the BOSSA Platform and, in general, developers of the Indoor Location Project, to take the baton and continue the work.

As future work, it could be an intriguing idea to offer the platform's resources in events such as hackathons, in which developers could play and experiment with the APIs and maybe suggest ideas for future developments. It would be enough with indicating them to access the website *api.iitrtclab.com*, where they could find all the needed information.
Additionally, it is suggested to continue the process for generation of new APIs and modules for the system, and to regularly update the documentation online.

Acronyms

PIDF-LO: Presence Information Data Format Location Object

BOSSA: BluetOoth and SenSors Array

APCO: Association of Public- Safety Communications Officials-International

NENA: National Emergency Numbers Association

API: Application Programming Interface

RSSI: Received Signal Strength Indicators

SIP: Session Initiation Protocol

VoIP: Voice over IP

RTP: Real-time Transport Protocol

PSAP: Public-Safety Answering Point

ESINet: Emergency Services IP Backbone Network

HTTP: Hypertext Transfer Protocol

CSS: Cascading Style Sheets

EJS: Embedded JavaScript

JS: JavaScript

RTC: Real-Time Communications

XML: eXtensible Markup Language

BLE: Bluetooth Low Energy

UUID: Universally Unique Identifier

AWS: Amazon Web Services

EC2: Elastic Compute Cloud

RDS: Relational Database Service

DB: Database

RDBMS: Relational Database Management System

DNS: Domain Name System

URL: Uniform Resource Locator

References

[1] Carol Davids, Cary Davids, Barat Ramaswamy Nandakumar, Neilabh Okhandiar, Francisco Rois, Chakib Ljazouli, Adrian Calle Murillo. “Dispatchable Indoor Location System for Mobile Phones based on a BlueTooth Low Energy Array”. Illinois Institute of Technology. Real-Time Communications Lab.

[2] Alberto González Trastoy. “A Dispatchable Bluetooth Indoor Location for Mobile Phones Calling for Emergency Services”. IIT July 2016.

[3] Francisco Jose Rois Siso. “New mobile application features: iterative location update”. December 2016

[4] Nathan Sowie. “Next deployment of the Indoor Location Project”.

[5] Adrian Calle Murillo “Indoor Location Project: New version of the NG911 Bluetooth User Application”. 2017

[6] Enzo Piacenza. “WebRTC Emergency Service Android APP”. Git repository: https://github.com/IIT-RTC-Lab/WebRTC_Emergency_Service_AndroidApp

[7] NG911 Bluetooth TestApp. Git repository: https://github.com/IIT-RTC-Lab/NG911-Bluetooth_TestApp

[8] Chakib Ljazouli. “Indoor Location Project – Database Design Document (DDD)”.

[9] C. Davids, J. M. Valdecantos, B. Dworak, C. Tovar, B. R. Nandakumar, and M. Patil, “Dispatchable indoor location for mobile phones calling for emergency services,” IPTComm ’15: Proceedings of the Principles, Systems and Applications on IP Telecommunications. ACM, October 2015.

[10] “Develop Web Apps in Node.js and Sails.js”. Mike McNeil, Founder at Sails.js. https://courses.platzi.com/courses/develop-apps-sails-js/.

[11] M. H. Dortch. (2014, November) “Roadmap for improving e911 location accuracy”. http://apps.fcc.gov/ecfs/document/view? id=60000986637

[12] “Apco.” https://www.apcointl.org/

[13] “Nena.” https://www.nena.org/

[14] “It real-time communications lab.” http://appliedtech.iit.edu/rtc- lab/

[15] “Detailed functional and interface standards for the nena i3 solution.” http://www.nena.org/Standards

[16] “Bluetooth core specification.”. https://www.bluetooth.com/specifications/bluetooth-core-specification

[17] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “Sip: Session initiation protocol. RFC 3261 (proposed standard).”

[18] “Webrtc.”. https://webrtc.org/

[19] http://www.systemware.com/sw_resource/analytics-2/ (04/20/2017)

[20] MySQL database system. https://www.oracle.com/mysql/index.html

[21] What is an API? YouTube, MuleSoft Videos

[22] Sublime Text 2. Text editor. https://www.sublimetext.com

[23] Node.js. https://nodejs.org/es/

[24] Sails.js framework. http://sailsjs.com

[25]https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

[26] Express.js. https://expressjs.com

[27] Grunt, the JavaScript Task Runner. https://gruntjs.com.

[28] Heroku platform cloud services. https://www.heroku.com/platform

[29] Git, version control. https://git-scm.com

[30] GitHub. https://en.wikipedia.org/wiki/GitHub

[31] EJS - Embedded JavaScript. http://ejs.co

[32] Bootstrap, from Twitter. http://bootstrapdocs.com/v2.0.2/docs/index.html

[33] https://www.npmjs.com

[34] http://sailsjs.com/documentation/concepts/deployment